Counting Bugs in Behavioural Models using Counterexample Analysis

Irman Faqrizal
Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG
38000 Grenoble, France

Gwen Salaün
Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG
38000 Grenoble, France

ABSTRACT

Designing and developing distributed software has always been a tedious and error-prone task, and the ever increasing software complexity is making matters even worse. Model checking automatically verifies that a model, e.g., a Labelled Transition System (LTS), obtained from higher-level specification languages satisfies a given temporal property. When the model violates the property, the model checker returns a counterexample, but this counterexample does not precisely identify the source of the bug. In this work, we propose some techniques for simplifying the debugging of these models. These techniques first extract from the whole behavioural model the part which does not satisfy the given property. In that model, we then detect specific states (called faulty states) where a choice is possible between executing a correct behaviour or falling into an erroneous part of the model. By using this model, we propose in this paper some techniques to count the number of bugs in the original specification. The core idea of the approach is to change the specification for some specific actions that may cause the property violation, and compare the model before and after modification to detect whether this potential bug is one real bug or not. Beyond introducing in details the solution, this paper also presents tool support and experiments.

KEYWORDS

Behavioural Models, Model Checking, Debugging, Counterexample, Bug Counting.

ACM Reference Format:
Irman Faqrizal and Gwen Salaün. 2022. Counting Bugs in Behavioural Models using Counterexample Analysis. In *International Conference on Formal Methods in Software Engineering (FormaliSE'22), May 18–22, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3524482.3527647

1 INTRODUCTION

Designing and developing distributed software has always been a tedious and error-prone task, and the ever increasing software complexity is making matters even worse. Model checking [1] is an established technique for automatically verifying that a model, e.g., a Labelled Transition System (LTS), obtained from higher-level specification languages such as process algebra satisfies a given temporal property, e.g., the absence of deadlocks. 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 (even thousands) of actions, (ii) the debugging task is mostly achieved manually, (iii) the counterexample does not explicitly highlight the source of the bug that is hidden in the model, (iv) the counterexample describes only one occurrence of the bug and does not give a global view of the problem with all its occurrences.

In this work, we rely on some debugging techniques proposed in [4]. These techniques aim at extracting the part of the whole behavioural model which does not satisfy the given property. This model is called a *counterexample LTS*. In that LTS, some specific states are identified from which the specification can reach a correct part of the model or an incorrect one. These *faulty states* correspond to decision points or choices that are particularly interesting because they usually point out a part of the model that may identify the source of the bug. Once all the faulty states have been identified, visualization techniques are used to graphically observe the whole model and see how those states are distributed over that model.

One limit of this approach is that it does not give any clue on how much erroneous is the specification and corresponding model. We do not know if there is one bug in the application or several ones. This information is important to quantify the faulty part of the model and thus to measure the effort for debugging the program. Moreover, bugs are usually considered to be all identical in terms of importance. However, this is not the case in practice, and some of them are more critical than others. As an example, a bug can only occur in a very specific situation whereas another one can systematically occur and thus make the whole program erroneous.

In this paper, we propose some techniques to count the number of bugs in a given specification. This approach relies on the idea of modifying the specification and of analyzing the changes in the corresponding model. As a result, we provide as output the number of bugs in the specification with some information regarding the kind of bugs. This information serves to quantify the degree of faultiness of the specification and can be used as a measure to evaluate the effort necessary to correct the specification. In this work, we focus on safety properties, which state that something wrong must not happen. These properties are widely used in the verification of real-world systems. The violation of a safety property occurs when the specification contains certain sequences of actions invalidating the property. We will show in this work how we take advantage of the last actions in these sequences to identify and count the number of bugs.

FormaliSE'22, May 18–22, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *International Conference on Formal Methods in Software Engineering (FormaliSE'22), May 18–22, 2022, Pittsburgh, PA, USA*, https://doi.org/10.1145/3524482.3527647.

More precisely, our approach takes as input a specification that can be compiled into an LTS (or directly an LTS) and a safety property written using a temporal logic. In this work, we use LNT [9] and MCL [21], respectively, as specification language and temporal logic. We first apply some successive changes to the specification to highlight some actions (known as last actions in the property) that may be the cause of bugs. Then, every time the specification is modified, we generate the corresponding counterexample LTS. By comparing the successive versions of the counterexample LTS, we are able to detect whether each potential bug is really a bug or not. At the end, we can return the number of bugs and for each bug we indicate whether it is inevitable or avoidable. This approach is fully automated by a tool we implemented and applied on several examples for validation purposes.

The rest of this paper is organized as follows. Section 2 introduces some preliminary notions on behavioural models and faulty states. Section 3 presents the details of our approach for bug counting. Section 4 introduces the tool support and some experiments we carried out for validating our approach. Section 5 surveys related work and Section 6 concludes the paper.

2 BACKGROUND

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 2.1. (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, and $T \subseteq S \times \Sigma \times S$ 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 [9] and compilers from the CADP toolbox [15] for obtaining LTSs from LNT specifications. However, the approach presented in this section is generic in the sense that it applies on LTSs produced from any specification language and any compiler/verification tool.

LNT is an extension of LOTOS [17], an ISO standardized process algebra, which allows the definition of data types, functions, and processes. Table 1 provides an excerpt of the behavioural fragment of LNT syntax. B stands for a LNT term, A for an action, E for a Boolean expression, x for a variable, T for a type, and P for a process name. The syntax fragment presented in this table contains the termination construct (**stop**) and the occurrence of actions (A) that may come with offers (send an expression E or receive in a variable x). LNT processes are then built using several operators: sequential composition (**;**), conditional statement (**if**), non-deterministic choice (**select**), parallel composition (**par**) where the communication between the involved processes is carried out by rendezvous on a list of synchronized actions, looping behaviours described using process calls or explicit operators (**while**), and assignment (**:=**) where the variable should be defined beforehand (**var**). LNT is formally defined using operational semantics based on LTSs.

B	::=	**stop**
	|	A (!E, ?x)
	|	B_1; B_2
	|	**if** E **then** B_1 **else** B_2 **end if**
	|	**select** B_1[]...[]B_n **end select**
	|	**par** $A_1, ..., A_m$ **in** B_1||...||B_n **end par**
	|	$P[A_1, ..., A_m](E_1, ..., E_n)$
	|	**while** E **loop** B **end loop**
	|	**var** x:T **in** x := E; B **end var**

Table 1: Excerpt of LNT Syntax (Behaviour Part)

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

Definition 2.2. (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 $t(M)$.

Model checking consists in verifying that an LTS model satisfies a given temporal property P, 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 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 formalized using a temporal logic. We use MCL (Model Checking Language) [21] in this work, which is an action-based, branching-time temporal logic suitable for expressing properties of concurrent systems. MCL is an extension of alternation-free μ-calculus with regular expressions, data-based constructs, and fairness operators. A safety property can be semantically characterized by an infinite set of traces t_P, corresponding to the traces that violate the property P in an LTS. If the LTS model does not satisfy the property, the model checker returns a *counterexample*, which is one of the traces characterized by t_P.

Definition 2.3. (Counterexample) Given an LTS $M = (S, s^0, \Sigma, T)$ and a property P, a counterexample is any trace which belongs to $t(M) \cap t_P$.

The new contributions of this paper rely on existing results presented in [4]. This approach takes as input a specification and a temporal property, and identifies decision points where the specification (and the corresponding LTS model) goes from a (potentially) correct behaviour to an incorrect one. These choices turn out to be very useful to understand the source of the bug. These decision points are called *faulty states* in the LTS model.

In order to detect these faulty states, the transitions in the model first need to be categorized into different types. The type of a transition indicates whether that transition belongs to traces satisfying the property or not. More precisely, transitions in the LTS can be categorized into three types:

- A *correct transition* belongs to traces of the LTS for which the property is satisfied.
- An *incorrect transition* belongs to traces of the LTS for which the property is violated.

- A *neutral transition* is a transition where the validity of the property is not yet determined.

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

Definition 2.4. (Tagged LTS) Given an LTS $M = (S, s^0, \Sigma, T)$, and the set of transition tags Γ, the tagged LTS is a tuple $M_T = (S_T, s^0_T, \Sigma_T, T_T)$ where $S_T = S$, $s^0_T = s^0$, $\Sigma_T = \Sigma$, and $T_T \subseteq S_T \times \Sigma_T \times \Gamma \times S_T$.

It is worth noting that a specific action appearing just once in the LNT specification can appear several times in the corresponding tagged LTS possibly with a different type. This is the case for instance when we have a choice between A and B, both followed by C (*select A [] B end select ; C* in LNT), and the property states that we do not want A and C in sequence. In that case, the sequence A followed by C corresponds to two incorrect transitions in the tagged LTS whereas the sequence B followed by C corresponds to two correct transitions.

The tagged LTS where transitions have been typed allows us to identify faulty states in which an incoming neutral transition is followed by a choice between at least two transitions with different types (correct, incorrect, neutral). Such a faulty state consists of all the neutral incoming transitions and all the outgoing transitions.

Definition 2.5. (Faulty State) Given a tagged LTS $M_T = (S_T, s^0_T, \Sigma_T, T_T)$, a state $s \in S_T$ is a faulty state if $\exists t = s' \xrightarrow{(l,\gamma)} s \in T_T$ where t is a neutral transition, and $\exists t_1 = s \xrightarrow{(l_1,\gamma_1)} s_1 \in T_T$ and $\exists t_2 = s \xrightarrow{(l_2,\gamma_2)} s_2 \in T_T$ where t_1 and t_2 are neutral, correct or incorrect transition, with different tags ($\gamma_1 \neq \gamma_2$). In that case, the faulty state s consists of 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 $t' = s \xrightarrow{(l,\gamma)} s'' \in T_T$.

By looking at outgoing transitions of a faulty state, four kinds of faulty states can be identified (Figure 1 (c)):

(1) with at least one correct transition and one neutral transition (no incorrect transition),
(2) with at least one incorrect transition and one neutral transition (no correct transition),
(3) with at least one correct and one incorrect transition (no neutral transition), and
(4) with at least one correct, one incorrect, and one neutral transition.

Finally, the notion of counterexample LTS is defined and will be used in the rest of this paper.

Definition 2.6. (Counterexample LTS) Given a tagged LTS $M_T = (S_T, s^0_T, \Sigma_T, T_T)$ and the set of faulty states F computed on this tagged LTS, the corresponding counterexample LTS is the tuple $C = (M_T, F)$.

The reader interested in more details regarding the algorithms for computing tagged and counterexample LTSs can refer to [4].

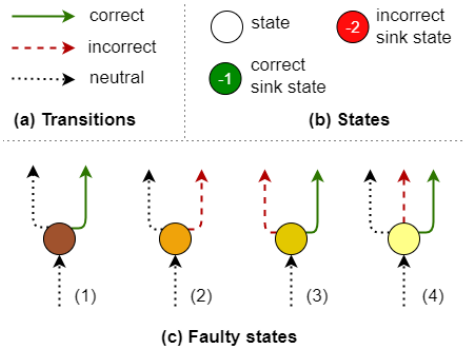

Figure 1: Legend for Transitions, Faulty States, and States

Counterexample LTSs can be visualized to give to the developer a graphical representation of these LTS models where correct/incorrect/neutral transitions and faulty states are highlighted. These visualization techniques make use of different styles and colours to distinguish the different types of transitions and states, as defined in Figure 1. There are three types of transitions (Figure 1 (a)): neutral transitions are represented using (black) dotted lines, correct transitions are represented using (green) solid lines, and incorrect transitions are represented using (red) dashed lines. As for states (Figure 1 (b)), white is used for regular states, (red) solid color with -2 as the state's identifier is used for incorrect sink states, and (green) solid color with -1 as the state's identifier is used for correct sink states. When a state is green or red, this is a sink state because from here the remaining states will be of the same color, so it is not necessary to keep that part of the LTS. Faulty states are represented with different shades of yellow depending on the type of the outgoing transitions (as shown in Figure 1 (c)).

Example. An example of LNT specification is presented in Listing 1. Suppose the property to be respected by this specification states that B should not be executed before A. This is formalized in MCL as follows:

$$[\ (not\ A) *\ .\ B\ ]\ false \tag{1}$$

```
1  process main [INIT1, INIT2, EXEC1,
2  EXEC2, EXEC3, A, B : any] is
3    INIT1 ;
4    select B ; A [] EXEC1 end select ;
5    select
6      A
7    []
8      INIT2 ;
9      select
10       par EXEC2 ; A || B end par
11     []
12       EXEC3
13   end select
14   end select
15 end process
```

Listing 1: Example of LNT Specification

By looking at the specification, we can see that the property can be violated in several ways. For instance, in the parallel statement at line 10, there is a possibility to execute B followed by *EXEC2* then A, or *EXEC2* followed by B then A. Both sequences violate the property because B is executed before A.

The counterexample LTS generated from this specification and property is given in Figure 2. The counterexample LTS is an LTS composed of all possible counterexamples. For instance, in the case of the property's violation at line 10, one of the corresponding counterexamples is: *INIT1* (from state 0), *EXEC1* (from state 1), *INIT2* (from state 2), *EXEC2* (from state 3), *B* (from state 4). This counterexample LTS exhibits all types of faulty states. As an example, the faulty state identified by 3 is a faulty state with neutral, correct, and incorrect transitions. From this state, the property's violation can be temporarily avoided by executing *EXEC2* (*neutral* transition), can be definitely avoided by executing *EXEC3* (*correct* transition), or can be violated by executing *B* (*incorrect* transition). Some states and transitions are hidden in the counterexample LTS because they are replaced by the (correct or incorrect) sink states. For example, in state 1 by executing *B*, the transition goes to an incorrect sink state (state -2). In that case, since the property becomes false, it is not useful to show any of the following states and transitions.

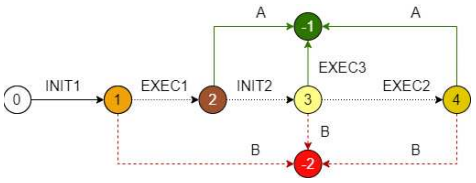

Figure 2: Example of Counterexample LTS

3 COUNTING BUGS

Given a specification (written in LNT) and a temporal property (written in MCL), we propose a method for counting the number of bugs. These techniques rely on several notions we will successively present in this section. First, we define the notion of last actions in property, which are the actions executed just before the property becomes false. We also define two different types of bugs called inevitable and avoidable bugs. When a bug is detected, the solution presented in this section is also able to indicate whether the bug is inevitable or avoidable. The techniques for counting bugs rely on the notion of *dummy choice*, which aims at replacing in the specification an occurrence of a last action in property by a choice between this action and a dummy action. This dummy choice is crucial to identify whether this action corresponds to a part of the specification where the property is violated or not. By relying on this notion of dummy choice, we present an algorithm, which computes the number of bugs and indicates for each bug whether it is inevitable or avoidable. This section ends with the generation of more precise counterexamples, by taking advantage of the information computed when counting the number of bugs.

3.1 Last Actions in Property

In this paper, we focus on safety properties. The main idea is that the last actions in the property are the actions executed just before the property becomes false. Therefore, these actions play a central role in the computation of the truth value of the property. In this work, these properties are written in MCL and have the following form: $[\ R\ ]\ false$, where R is a regular expression. The syntax of regular expressions is given by:

$$R\ ::=\ true \mid a \mid \neg a \mid a_1 \vee a_2 \mid R1.R2 \mid R1|R2 \mid R1*$$

where $a\ \in\ \Sigma$ for a given LTS $M = (S, s^0, \Sigma, T)$. We assume that the regular expression is not empty, therefore the set of last actions in property is not empty either and is computed by a function called $last$. Note that $\vee$ and $|$ are slightly different because $\vee$ applies on actions whereas $|$ applies on formulas. There is no $\wedge$ operator in the syntax because this predicate results in a property that always evaluates to true [1].

Definition 3.1. (Last Actions in Property) Given a safety property $[\ R\ ]\ false$, function $last$ takes R and Σ as inputs and returns the set of last actions in this property. Function $last$ is defined as follows:

$$\begin{aligned}
&last(true,\ \Sigma) && = \Sigma\\
&last(a,\ \Sigma) && = \{a\}\\
&last(\neg a,\ \Sigma) && = \Sigma \setminus \{a\}\\
&last(\ a_1 \vee a_2,\ \Sigma) && = \{a_1, a_2\}\\
&last(R1.R2,\ \Sigma) && = if\ empty(R2)\ then\ last(R1,\ \Sigma)\ else\\
& && \quad last(R2,\ \Sigma)\ end\ if\\
&last(R1|R2,\ \Sigma) && = last(R1,\ \Sigma)\ \cup\ last(R2,\ \Sigma)\\
&last(R1*,\ \Sigma) && = \emptyset
\end{aligned}$$

where function $empty$ takes as input a regular expression and returns $true$ if this expression is the empty word and $false$ otherwise.

Example. We illustrate the computation of last actions in a property with several properties. First, consider the following property, stating that action *CLOSE* should not be executed after action *EXEC*. This property is written in MCL as follows:

$$[true*\ .\ EXEC\ .\ true*\ .\ CLOSE]\ false \tag{2}$$

In this property, action *CLOSE* is the last action in property because it is the only action which is executed just before the property becomes false. Consider now a property stating that when *EXEC* is executed then *CLOSE* must not be executed right after it:

$$[true*\ .\ EXEC\ .\ \neg\ CLOSE]\ false \tag{3}$$

In this case, the last actions in property is the set of all actions in the alphabet of the LTS model except *CLOSE*, that is, $\Sigma \setminus \{CLOSE\}$. This example shows that the last action in property is not always the rightmost action written in the property. Let us assume finally that we do not want the execution of a sequence of *OPEN* or *CLOSE* actions, which is written in MCL as follows:

$$[true*\ .\ (OPEN\ .\ OPEN) \mid true*\ .\ (CLOSE\ .\ CLOSE)]\ false \tag{4}$$

The resulting set of last actions consists of two actions { *OPEN*, *CLOSE* }. In the rest of this paper, we will assume that the property has a unique last action for simplifying the presentation, but a property can have several last actions as illustrated previously.

[1] Assume a property defined as follows: $[A.\ (B\ \wedge\ C)]\ false$. This property is true on all LTS models. This comes from the $B\ \wedge\ C$ predicate, which is always false since B and C are different actions names (i.e., no label of an LTS can satisfy B and C). Since the predicate $B\ \wedge\ C$ is false, the regular formula $A.\ (B\ \wedge\ C)$ does not match any transition sequence in any LTS, so the modality (which expresses the absence of such sequences) always evaluates to true.

3.2 Inevitable and Avoidable Bugs

A bug is called an *inevitable bug* when the property's violation can not be avoided from the initial state of the LTS model. In that case, there is no trace in the model satisfying the property. In contrast, an *avoidable bug* is a bug where the violation of the property can be avoided. This means that there exists at least a trace in the LTS model for which the property is not violated.

Definition 3.2. (Inevitable Bug) Given an LTS $M = (S, s^0, \Sigma, T)$ and a property P, there is an inevitable bug in M if $\nexists\, t \in t(M)$ such that $t \models P$.

Definition 3.3. (Avoidable Bug) Given an LTS $M = (S, s^0, \Sigma, T)$ and a property P, there is an avoidable bug in M if $\exists\, t, t' \in t(M)$ such that $t \models P$ and $t' \not\models P$.

Example. We illustrate this with the LNT specifications given in Listings 2 and 3. The property we want to verify on these specifications was introduced in Section 3.1 and states that action *CLOSE* should not be executed after action *EXEC*. In Listing 2, there is an inevitable bug because *CLOSE* (line 10) is always executed after *EXEC* (line 4). In Listing 3, there is a possibility to execute *CLOSE* (line 9) after *EXEC* (line 4) is executed. However, it is also possible to execute the first part of the select statement (line 6) and thus avoid the property violation. This means that the bug is avoidable.

```
1  process Main [INIT, EXEC,
2    CLOSE, AA, BB, CC: any] is
3      INIT;
4      EXEC;
5      select
6          AA; BB
7      []
8          CC
9      end select;
10     CLOSE
11 end process
```

Listing 2: Inevitable Bug

```
1  process Main [INIT, EXEC,
2    CLOSE, AA, BB, CC: any] is
3      INIT;
4      EXEC;
5      select
6          AA; BB
7      []
8          CC;
9          CLOSE
10     end select
11 end process
```

Listing 3: Avoidable Bug

The counterexample LTS corresponding to the LNT specification given in Listing 2 is shown in Figure 3 (a). It has all incorrect (red) transitions because the property violation is inevitable from the initial state. In contrast, the counterexample LTS for Listing 3 (Figure 3 (b)) exhibits one faulty state (state 2): if *AA* is executed, the bug is avoided, whereas if *CC* is executed, the property is violated. Note that in this second counterexample LTS, all states and transitions after *AA* are hidden and replaced by a correct sink state (i.e., *BB* is missing) because it is not possible to violate the property after *AA* is executed.

3.3 Dummy Choices

The techniques for counting bugs rely on the notion of *dummy choice*. Given an action, the main idea is to replace the occurrence of this action by a choice in the specification between executing this action and executing a fresh action called *dummy action*.

Definition 3.4. (Dummy Choice) Given a specification L and an action A appearing in L, the addition of a dummy choice transforms A into a choice statement (select construct in LNT) with two clauses: the first clause contains the action A and the second clause contains a new action $DUMMY$ which is not used anywhere in the specification L.

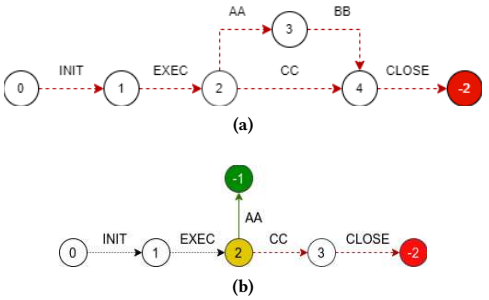

Figure 3: Counterexample LTSs for Specifications in Listings 2 (a) and 3 (b)

For debugging purposes, the action replaced by a dummy choice corresponds to the last action in property. Indeed, the property violation occurs when the last action in property is executed. Our method aims at adding a new choice in the specification in order to avoid this execution. This dummy choice helps to identify whether this specific occurrence of the last action may cause the violation of the property (real bug) or not. To do so, we need two versions of the specification: a version where the dummy choice has been added and the version where that dummy choice has been removed. For each version of the LNT specification, we generate the corresponding counterexample LTS. As a result, we have two counterexample LTSs and we compare them in order to decide whether this occurrence of the last action in property corresponds to a real bug or not. More precisely, this comparison has three possible outcomes:

(1) Transition(s) labelled with dummy action(s) are missing from non-faulty state(s) in the second counterexample LTS. In this case, the occurrence of the last action in property **does not correspond to a bug**.
(2) At least one transition labelled with a dummy action is missing from a faulty state in the second counterexample LTS. This means that the occurrence of the last action in property corresponds to an **avoidable bug**.
(3) The counterexample LTS changed to a counterexample LTS with all incorrect transitions. This means that the occurrence of the last action in property corresponds to an **inevitable bug**.

These three cases are written more formally in the following definition.

Definition 3.5. (Counterexample LTS Comparison) Given two counterexample LTSs $C = ((S_T, s_T^0, \Sigma_T, T_T), F)$ and $C' = ((S'_T, s'^0_T, \Sigma'_T, T'_T), F')$ generated from two specifications L (with a dummy choice) and L' (without that dummy choice), the three possible outcomes obtained by comparing C and C' are as follows:

(1) No bug: $\forall t = s \xrightarrow{(DUMMY,\gamma)} s' \in T,\ s \notin F,\ and\ t \notin T'$
(2) Avoidable bug: $\exists t = s \xrightarrow{(DUMMY,\gamma)} s' \in T,\ s \in F,\ and\ t \notin T'$
(3) Inevitable bug: $\exists s \xrightarrow{(A,correct)} s' \in T,\ and\ \forall s'' \xrightarrow{(A',\gamma')} s''' \in T',\ \gamma' = incorrect$

Note that these three cases are exclusive and cover all possible cases. By removing a dummy choice, the counterexample LTS entirely change (case (3), this is an inevitable bug) or not. If this is not an inevitable bug, there are only two cases: (i) removing the dummy choice has no real impact in the sense that there is just one less transition (case (1)), or (ii) removing the dummy choice has an impact, which results in the presence of a new faulty state in the corresponding counterexample LTS.

Example. Consider the LNT specifications given in Listings 2 and 3. The last action in property consists of the action *CLOSE*. Therefore, for both examples, our technique replaces the actions *CLOSE* at line 10 in Listing 2 and line 9 in Listing 3 by the following select statement:

select CLOSE [] *DUMMY end select*

The corresponding counterexample LTSs are given in Figure 4 (a) and (b). In the first counterexample LTS (Figure 4 (a)), there is a faulty state that has an outgoing transition with a dummy action. If this dummy choice is removed and the counterexample LTS generated, the resulting counterexample LTS consists of incorrect transitions only (Figure 3 (a)). As a result, we can conclude that this occurrence of last action in property corresponds to an inevitable bug (case 3). In the second counterexample LTS (Figure 4 (b)), if the dummy choice is removed in the specification, a dummy action is missing from a faulty state in the generated counterexample LTS (Figure 3 (b)). This means that the corresponding bug is classified as an avoidable bug (case 2).

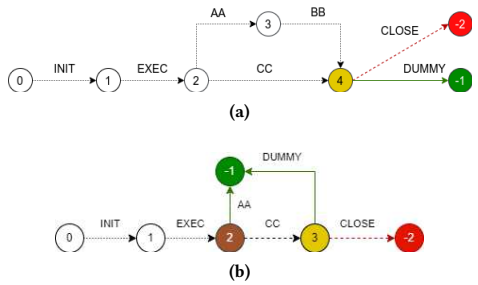

Figure 4: Counterexample LTSs with Dummy Choices

3.4 Algorithm

Before presenting the algorithm computing the number of bugs, we would like to clarify what we mean by *one bug* and how we count them. Indeed, when the specification does not satisfy a given property, there might be several ways to correct the bug(s) in the corresponding specification. One solution, which induces minimum changes, is to correct the last action in property. In this work, we assume that one bug can be fixed by making a correction where the last action in property is located in the specification, by removing this action for instance. Therefore, the number of bugs in our approach coincides with the number of times one needs to correct the occurrences of the last action in property corresponding to real bugs.

Algorithm 1: Algorithm for Counting Bugs

Inputs : L (LNT specification), P (MCL property)
Outputs: $n_{avd}, n_{inv} \in \mathbb{N}$

1 — Step 1: addition of dummy choices —
2 $(S, s^0, \Sigma, T) := generateLTS(L)$;
3 $R := getRegExpr(P)$;
4 $A := last(R,\ \Sigma)$;
5 $D := \emptyset$;
6 $counter := 1$;
7 **foreach** *action in* $getActions(L)$ **do**
8 **if** $action \in A$ **then**
9 $dummy :=$ "dummy" $+\ counter$;
10 $L := addDummyChoice(action,\ dummy,\ L)$;
11 $D := D \cup \{dummy\}$;
12 $counter := counter + 1$;
13 **end**
14 $(M_T, F) := generateCLTS(L,\ P)$;
15 — Step 2: identification of dummy choices corresponding to real bugs —
16 $D' := \emptyset$;
17 **foreach** s *in* $getStates(M_T)$ **do**
18 **if** $(\exists t = s \xrightarrow{(l,\gamma)} s' \in getTrans(M_T)\ s.t.\ l \in D)\ and\ (s \in F)$ **then**
19 $D' := D' \cup \{getDummyAction(s)\}$;
20 **end**
21 — Step 3: bug counting for each kind of bug —
22 $n_{avd} := 0$;
23 $n_{inv} := 0$;
24 **foreach** *dummy in* D' **do**
25 $L' := L$;
26 $L' := removeDummyChoice(dummy,\ L')$;
27 $(M'_T, F') := generateCLTS(L',\ P)$;
28 **if** $F' \neq \emptyset$ **then**
29 $n_{avd} := n_{avd} + 1$;
30 **if** $F' = \emptyset$ **then**
31 $n_{inv} := n_{inv} + 1$;
32 **end**
33 return n_{avd}, n_{inv};

We now present Algorithm 1, which takes as input an LNT specification and an MCL property, and returns the number of inevitable and avoidable bugs. The main idea of this algorithm is that we first replace all occurrences of the last action in property by dummy choices. In the next step, by removing the dummy choices one by one and comparing the resulting counterexample LTSs (before and after removal of each dummy choice), we can deduce whether this specific occurrence of last action corresponds to a bug, and if this is the case to which kind of bug.

More precisely, in step 1, the algorithm adds the dummy choices to all occurrences of the last action in property (lines 7 to 13). The corresponding counterexample LTS is then generated (line 14). In

step 2, the algorithm loops over all the states in the counterexample LTS. When it finds a dummy action belonging to a transition outgoing from a faulty state, then that dummy action is added to the set of dummy actions D' (lines 17 to 20). This second step is useful to find all occurrences of the last action in property which correspond to real bugs (case 2 and 3 in Section 3.3).

In the last step, the algorithm loops over the set of remaining dummy actions (line 24). In each iteration, a dummy choice (corresponding to a dummy action in the set) is removed from a copy of the LNT specification, then the counterexample LTS is generated from it (lines 25 to 27). If the counterexample does not have any faulty state, then the number of inevitable bugs is incremented. Otherwise, the number of avoidable bugs is incremented (lines 28 to 31).

As far as the complexity of the algorithm is concerned, the most costly step is to generate counterexample LTSs, which is computed a first time (line 14) and then n times where n is the number of dummy actions in the set of dummy actions D'. The counterexample LTS is obtained by first compiling the LNT specification to LTS, and then by computing a product between this LTS and the LTS obtained from the MCL formula. The compilation of LNT to LTS relies on a transformation to Petri nets, whose complexity (exponential) is the same as computing the corresponding marking graph [12]. In practice, it is less expensive as parts of the net are 1-bounded. As for the product, the complexity is $O(S * S_F)$ where S is the number of states of the initial LTS and S_F is the number of states of the formula LTS. As a result, the overall complexity resides in the repetition $n+1$ times of the execution of an exponential algorithm for computing a counterexample LTS.

Last but not least, to go further and actually debug the original specification, we rely on the inevitable bugs to compute what we call the *final counterexample LTS*. In this LTS, we remove dummy choices for actions that are not bugs or avoidable bugs, but we keep dummy choices for inevitable bugs. This allows us to generate a counterexample with faulty states for all bugs (inevitable and avoidable). We will detail in Section 3.5 how we can take advantage of this final counterexample LTS to compute more precise counterexamples for debugging purposes.

```
1  process Main [INIT, EXEC, CLOSE, AA, BB, CC: any] is
2    INIT;
3    select
4      AA; EXEC; BB; CLOSE        (* 1st (avoidable) bug *)
5    []
6          CLOSE; CC              (* Not a bug *)
7    []
8          par
9            CLOSE                (* 2nd (avoidable) bug *)
10         ||
11           EXEC
12         end par
13   end select;
14   EXEC; CLOSE                  (* 3rd (inevitable) bug *)
15 end process
```

Listing 4: LNT Specification with Several Bugs

Example. Listing 4 gives an example of LNT specification on which we want to check the property introduced beforehand: *CLOSE* must not be executed after *EXEC*. There are three bugs in this example. The first one occurs if we execute the first part of the select statement (line 4). The second one occurs when we execute the third part of the select statement (lines 8 to 12). The last one is an inevitable bug and appears at the end of the specification (line 14).

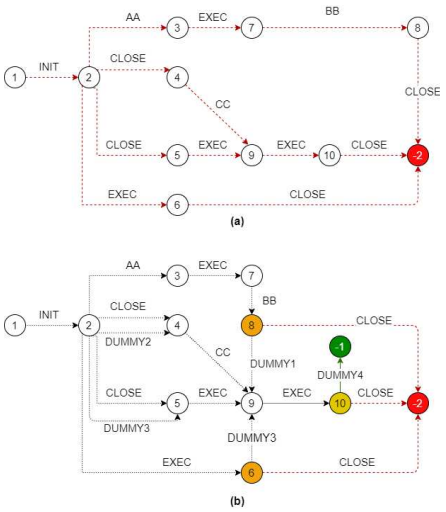

Figure 5: Counterexample LTS before (a) and after (b) Addition of Dummy Choices

The first step of the algorithm is to add dummy choices to all occurrences of the last action in property in the LNT specification. Note that since we increment the dummy action suffix for each addition of a dummy choice, each new dummy action is unique (e.g., in Listing 4, the dummy action in the first dummy select at line 4 is *DUMMY1*, it is *DUMMY2* at line 6, etc.). Figure 5 (a) shows the counterexample LTS generated from the original specification. In that counterexample LTS, all transitions are incorrect because there is one inevitable bug. Figure 5 (b) shows the counterexample LTS corresponding to the specification where each occurrence of the last action in property has been replaced with a dummy choice.

In the next step of Algorithm 1, the counterexample LTS is traversed (Figure 5 (b)) to check if there is any dummy action belonging to transitions outgoing from faulty states. There are three faulty states (states 6, 8, and 10), and each of them has an outgoing transition with a dummy action. Therefore, these dummy actions correspond to real bugs and we add them to the set of dummy actions ($\{DUMMY1, DUMMY3, DUMMY4\}$).

The last step loops over the set of dummy actions. In each iteration, we remove a dummy choice (from the LNT specification) corresponding to a dummy action in the set, and generate the counterexample LTS. Figure 6 shows the corresponding counterexample LTSs. The counterexample LTS (a) and (b) are the counterexample LTSs generated after the first and second iterations (i.e., removal of the *DUMMY1* and *DUMMY3* actions). In these two iterations, both generated counterexample LTSs still have faulty states. This implies that these occurrences of the last action in property correspond to avoidable bugs. As for the last iteration, the resulting counterexample LTS (c) is entirely incorrect and thus has no faulty states. This means that this particular occurrence of the last action in property corresponds to an inevitable bug. The algorithm finally

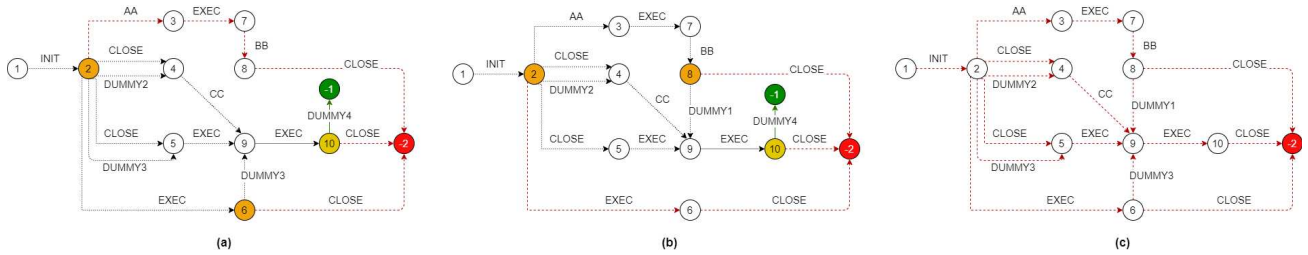

Figure 6: Iterations in the Final Step of the Bug Quantification Algorithm

returns that this LNT specification exhibits two avoidable bugs and one inevitable bug.

3.5 Counterexamples

In this final part of the section, we show how we can generate counterexamples from a counterexample LTS. This makes particulary sense from the final counterexample LTS introduced in Section 3.4. Counterexamples are a classic and simple way to represent a single execution leading to a state where the property is violated. Here, we generate more precise counterexamples because these traces also contain colored transitions and faulty states. Since a counterexample LTS is composed of all possible counterexamples, there are several options for generating counterexamples from such models. Therefore, we propose a set of eleven strategies for generating these counterexamples categorized in four groups: (i) classic counterexamples (e.g., random or shortest counterexample), (ii) counterexamples based on faulty states (e.g., shortest path to a faulty state then shortest counterexample or counterexample with the fewest number of faulty states), (iii) probability-based counterexamples (e.g., the counterexample with the highest probability[2]), (iv) counterexamples based on the distance to the bug (e.g., the counterexample including the faulty state closest to the incorrect sink state, possibly with the fewest number of faulty states).

For the sake of space, we cannot precisely present all these strategies, but we will present some of them on concrete examples in the rest of this section. The reader can find more details for each strategy in [13] as well as guidelines explaining how to choose which strategy to use. It is worth emphasizing that the counterexamples generated here are not classic traces but they also include colors for transitions and (faulty) states as well as all outgoing transitions for faulty states. This information is particularly helpful for debugging purposes.

Example. Consider the counterexample LTS presented in Figure 7, which is the final counterexample LTS after application of Algorithm 1 on the LNT specification given in Listing 4. This is the final counterexample LTS where a single dummy action (*DUMMY4*) corresponding to the inevitable bug is preserved in this model.

Figure 8 shows a first example of counterexample generated using a classic strategy (shortest counterexample). These counterexamples are usually convenient because they exhibit the shortest trace causing the property's violation. In this case, the sequence of actions is *INIT*, *EXEC*, *CLOSE*. We can deduce from the generated

[2]We assume by default that all transitions have equiprobable probabilities.

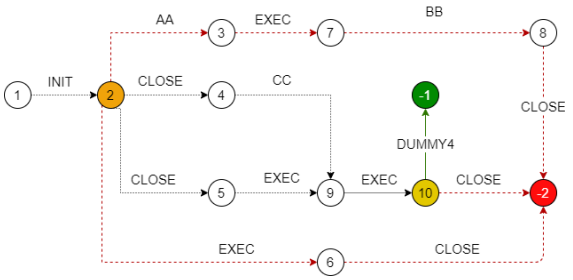

Figure 7: Final Counterexample LTS

counterexample that after we execute EXEC (second action) the bug becomes inevitable. Moreover, by focusing on the faulty state 2, we can see that there is a possibility to avoid the property's violation by executing *CLOSE*.

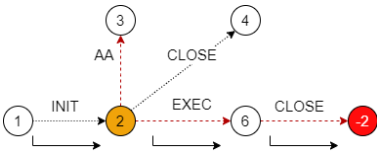

Figure 8: Shortest Counterexample

Figure 10 shows another example of counterexample, which was computed in that case using a strategy based on the distance to the bug, namely, the shortest counterexample with a faulty state closest to the incorrect sink state. This counterexample exhibits more faulty states than the former one. The first faulty state shows that *CLOSE* is the only option to avoid the bug. The second faulty state includes a dummy action, showing that this bug is inevitable, and the execution of action *CLOSE* leads to the property's violation.

4 TOOL SUPPORT

In this section, we present the tool support and some experiments we carried out for validating our approach.

4.1 Implementation

We have developed a tool in Java which implements the techniques presented in this paper to count the number of bugs. This tool relies on the CLEAR tool [3] for computing counterexample LTSs. Figure 9 gives an overview of the tool support. First of all, the last actions in

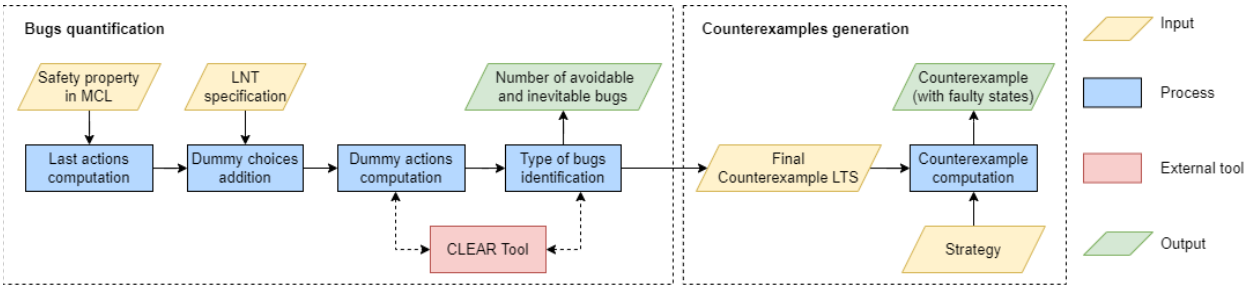

Figure 9: Tool Support

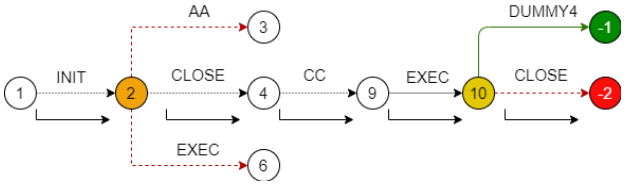

Figure 10: Shortest Counterexample with a Faulty State Closest to the Incorrect Sink State

property are computed, followed by the addition of dummy choices to the LNT specification. The tool then locates in the corresponding counterexample LTS the dummy actions which correspond to the property violation. For each dummy action, it identifies the type of bugs (i.e., inevitable or avoidable) by comparing the counterexample LTSs before and after removal of that specific dummy choice. As a result, the tool returns the number of inevitable and avoidable bugs. Finally, the user can generate some specific counterexamples using one of the proposed strategies.

4.2 Experiments

In this section, we present some of the experiments we carried out for validating our approach. The goal of these experiments was to evaluate: (i) the accuracy of the results computed by our debugging techniques and (ii) the time it takes to compute these results. Table 2 summarizes some of these experiments on ten realistic case studies. The first three columns characterize the example by giving a name, the reference (when available), the number of lines in LNT, and the size of the corresponding LTS generated from the LNT specification using CADP compilers (S, T, and L, stand for the number of states, transitions, and labels, respectively). As far as outputs are concerned, the three following columns give the size of the final counterexample LTS (S, T, L, and the number of faulty states, FS), the number of inevitable and avoidable bugs, and the time it takes (in seconds) to apply all the steps of our algorithm to compute the expected result.

These experiments allowed us to check whether our results were accurate in terms of number of bugs. To evaluate this goal, we first identify the actual number of inevitable and avoidable bugs by analysis of the program by an independent expert (using classic model checking combined with manual inspection). Then, we compare this result with the output of the tool support (fifth column in Table 2). As a result, both results coincided for all the examples on which these experiments were carried out.

Regarding performance, the objective was to show that it is possible to compute the number of bugs within a reasonable time. Consider for instance the example that takes the longest time to complete in the table, which is the Consumer-Producer example with 10 bugs (fourth row). It requires about three minutes to compute the result. This time mainly comes from the successive generation of the counterexample LTS (for each occurrence of last action in property), which requires the enumeration of all possible execution paths in the input specification. This time increases with the size of the LTS model and with the number of last action occurrences. It is worth mentioning that we do not aim in this paper at solving the state space explosion problem existing for model checking techniques. If we focus on a smaller example (trains stations, last row), it takes less than 30 seconds to compute the result because the LTS is smaller and there are fewer potential bugs in the original specification. Since the verification techniques proposed in this paper are applied at design time, these analysis times are not short but remain reasonable.

5 RELATED WORK

There are several works which focus on counterexample interpretation and comprehension, see, *e.g.* [2, 6, 16, 18, 19, 22]. In [6], 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]. CloSpan is also adopted in [19], 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 [6, 19] 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 [18], the authors propose a method to interpret counterexamples generated for liveness properties by dividing them into fated and free segments. Fated segments represent inevitability w.r.t. the failure, pointing out progress towards the bug, while free segments highlight the possibility to avoid the bug. The proposed approach classifies states in a state-based model in different layers (which

Example	LNT (LOC)	LTS (S/T/L)	Final CLTS (S/T/L/FS)	Nb of bugs (Inev/Avoid)	Time (s)
1. Work Stations	113	810 / 3771 / 25	483 / 1817 / 25 / 220	0 / 4	167
2. Consumer-Producer [5]	125	273 / 830 / 23	478 / 1381 / 20 / 62	1 / 3	78
3. Consumer-Producer (7 bugs)	128	327 / 1034 / 23	555 / 1574 / 22 / 116	2 / 5	162
4. Consumer-Producer (10 bugs)	131	381 / 1238 / 23	628 / 1832 / 23 / 170	3 / 7	196
5. Business Trip [20]	156	424 / 1232 / 13	25 / 63 / 12 / 6	0 / 1	61
6. RCC Mutex	204	48 / 66 / 15	20 / 29 / 15 / 16	0 / 1	24
7. IFTTT [14]	212	195 / 561 /23	316 / 851 / 21 / 19	1 / 2	60
8. Mutex [7]	284	783 / 1958 / 19	1553 / 3433 / 20 / 167	1 / 2	88
9. Mutex (8 bugs)	289	1437 / 4158 / 19	2775 / 6579 / 23 / 333	3 / 5	184
10. Train Stations [23]	357	59 / 98 / 18	34 / 57 / 16 / 8	0 / 1	25

Table 2: Experiments on Realistic Examples

represent distances from the bug) and produces a counterexample annotated with segments by exploring the model. 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 [16], 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. Similarly to our work, the work in [16] also wants to explain the counterexample with a focus on safety properties. In our work, we focus on a model containing all counterexamples, and we take advantage of that model to count the number of bugs. In contrast, their method relies on the analysis of a single counterexample and its variations, making sure that negative variations are from the same bug.

Delta debugging [25] is a well-known line of work on fault localization using testing techniques. The aim of delta debugging consists in understanding the minimal differences (deltas) between a successful and a failing program execution. There have been numerous published papers which rely on delta debugging, but, to the best of our knowledge, none of them allows the bug counting.

Several works aim at providing multiple counterexamples for facilitating debugging. Copty et al. [10] propose a counterexample wizard to help users when debugging applications using counterexamples. This wizard can retrieve multiple counterexamples, which can be used to understand and fix all reported errors for the given specification. In [8], the authors present a μ-calculus model checker that produces complete and complex counterexamples for branching logics like Computation Tree Logic. This framework also supports additional functionalities to translate these rich explanations back to the original logic. In [11], the authors present an approach relying on LTL model checking, which produces multiple counterexamples and which is able to reduce the number of counterexamples by detecting whether two counterexamples are equivalent. These approaches share similarities with our work because they focus on the generation of multiple counterexamples. Nevertheless, they do not exploit this result as we did for providing additional relevant information to developers such as the number of bugs or more precise counterexamples.

6 CONCLUDING REMARKS

In this paper, we have presented a solution for counting the number of bugs when analyzing a specification using model checking techniques. To do so, we focus on specific decision points in the specification, that make the corresponding model go to a correct or an incorrect portion of the model. These faulty choices are particularly helpful during the debugging task because they highlight the cause of the violation of a property. The approach relies on the counterexample LTS, which is an LTS consisting of all possible counterexamples. More specifically, we have proposed an algorithm to compute the number of bugs in the specification, which allows one to quantify how buggy is the specification and thus the effort required in the debugging phase. The main idea of this algorithm is to use some specific actions in the property to emphasize some parts of the specification that may cause its violation. Then, by comparing successive versions of the counterexample LTS, we manage to identify whether these suspicious pieces of the specification are actual bugs or not. Finally, the generation of counterexamples is possible from the final counterexample LTS. These counterexamples differ from regular ones because they include colored transitions and faulty states, and can be obtained using different strategies. The whole approach is fully automated by a tool we implemented and validated on several examples for validation purposes.

The main perspective of this work is to support liveness properties. In this case, counterexample LTSs and counterexamples have different shapes (lassos), and the approach thus deserves to be revisited, in particular the part comparing counterexamples LTSs.

REFERENCES

[1] C. Baier and J.-P. Katoen. 2008. *Principles of Model Checking.* MIT Press.
[2] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. 2003. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *Proc. of POPL'03.* ACM, 97–105.
[3] Gianluca Barbon, Vincent Leroy, and Gwen Salaün. 2019. Debugging of Behavioural Models with CLEAR. In *Proc. of TACAS'19 (LNCS, Vol. 11427).* Springer, 386–392.
[4] Gianluca Barbon, Vincent Leroy, and Gwen Salaün. 2021. Debugging of Behavioural Models using Counterexample Analysis. *IEEE Trans. Software Eng.* 47, 6 (2021), 1184–1197.

[5] Gianluca Barbon, Vincent Leroy, Gwen Salaün, and Emmanuel Yah. 2019. Visual Debugging of Behavioural Models. In *Proc. of ICSE'19*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 107–110.

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

[7] James E. Burns and Nancy A. Lynch. 1993. Bounds on Shared Memory for Mutual Exclusion. In *Information and Computation*, Vol. 107. Academic Press, 171–184.

[8] Simon Busard and Charles Pecheur. 2018. Producing Explanations for Rich Logics. In *Proc. of FM'18 (LNCS, Vol. 10951)*. Springer, 129–146.

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

[10] Fady Copty, Amitai Irron, Osnat Weissberg, Nathan P. Kropp, and Gila Kamhi. 2003. Efficient debugging in a formal verification environment. *Int. J. Softw. Tools Technol. Transf.* 4, 3 (2003), 335–348.

[11] Alma L. Juarez Dominguez and Nancy A. Day. 2013. *Generating Multiple Diverse Counterexamples for an EFSM.* Technical Report CS-2013-06. University of Waterloo.

[12] Javier Esparza. 1996. Decidability and Complexity of Petri Net Problems - An Introduction. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets (LNCS, Vol. 1491)*. Springer, 374–428.

[13] Irman Faqrizal. 2021. *Debugging and Quantifying Behavioural Models Using Counterexample Analysis.* Master's thesis. University Grenoble Alpes, France.

[14] Irman Faqrizal and Gwen Salaün. 2020. Clusters of Faulty States for Debugging Behavioural Models. In *Proc. of APSEC'20*. IEEE, 91–99. https://doi.org/10.1109/APSEC51365.2020.00017

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

[16] Alex Groce and Willem Visser. 2003. What Went Wrong: Explaining Counterexamples. In *Proc. of SPIN'03 (LNCS, Vol. 2648)*. Springer, 121–135.

[17] ISO. 1989. *LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* Technical Report 8807. ISO.

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

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

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

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

[22] Kavita Ravi and Fabio Somenzi. 2004. Minimal Assignments for Bounded Model Checking. In *Proc. of TACAS'04 (LNCS, Vol. 2988)*. Springer, 31–45.

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

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

[25] Andreas Zeller. 2009. *Why Programs Fail - A Guide to Systematic Debugging.* Academic Press.