Automated Repair of Violated Eventually Properties in Concurrent Programs

Irman Faqrizal, Quentin Nivon, Gwen Salaün

Automated Repair of Violated Eventually Properties in Concurrent Programs

Irman Faqrizal
Univ. Grenoble Alpes, CNRS,
Grenoble INP, Inria, LIG
38000 Grenoble, France
irman.faqrizal@inria.fr

Quentin Nivon
Univ. Grenoble Alpes, CNRS,
Grenoble INP, Inria, LIG
38000 Grenoble, France
quentin.nivon@inria.fr

Gwen Salaün
Univ. Grenoble Alpes, CNRS,
Grenoble INP, Inria, LIG
38000 Grenoble, France
gwen.salaun@inria.fr

ABSTRACT

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. Moreover, manually correcting the given specification or model can be a painful and complicated task. In this paper, we propose some techniques for computing patches that can correct an erroneous specification violating an eventually property. These techniques first extract from the whole behavioural model the part which does not satisfy the given property. In a second step, this erroneous part is analysed using several algorithms in order to compute the minimal number of patches in the specification so as to make it satisfy the given property. The approach is fully automated using a tool we implemented and applied on a series of examples for validation purposes.

ACM Reference Format:
Irman Faqrizal, Quentin Nivon, and Gwen Salaün. 2024. Automated Repair of Violated Eventually Properties in Concurrent Programs. In *Formal Methods in Software Engineering (FormaliSE) (FormaliSE '24), April 14–15, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3644033.3644383

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/-fog 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 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 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 have a specific focus on liveness properties, and more precisely on *single inevitable execution* properties also known as *eventually* or *existence* properties. Model checking books [2, 9, 10] present this property as one of the classic patterns of liveness properties. According to an empirical study carried out in [11] on more than 500 properties, this property is part of the 4-5 classic patterns of liveness properties most used by developers in practice.

In this paper, we consider concurrent programs or specifications that can transform into behavioural models such as Labelled Transition Systems (LTS). This is the case of most process algebraic specification languages such as CCS, CSP or LNT. Given an eventually property and such a specification or model, we rely on existing techniques [3, 4] to identify transitions that satisfy or violate the property in the model. This extended model, called Counterexample LTS (CLTS), contains important decision points called *faulty states* which can be used to identify bugs in the specification. However, debugging the specification using such a model is not straightforward especially when it consists of many states and transitions.

The main contribution of this paper is to propose techniques that can take advantage of the aforementioned CLTS for automatically finding and then optimally patching the bugs in the specification caused by the violation of an eventually property. More precisely, our approach first analyses the model in order to extract certain information associated with the property's violation. As a result, this analysis returns several sets of incorrect transitions. In a second step, we rely on these results for computing the minimal number of modifications to make on the specification such that the property is satisfied. A patch changes all the incorrect transitions into correct ones with (if any) the least impact on the correct parts of the specification. Our approach is implemented as a program that fully automates the computation of patches for avoiding the tedious, time-consuming and error-prone debugging steps that are usually achieved manually by users. This program was validated by performing several experiments on realistic specifications with their respective eventually properties.

The rest of this paper is organised as follows. Section 2 introduces preliminary notions. Section 3 presents the automated computation

FormaliSE '24, April 14–15, 2024, Lisbon, Portugal

of patches. Section 4 introduces tool support and experimental results. Section 5 surveys related work and Section 6 concludes.

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 actions, 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 $l \in \Sigma$ and $s, s' \in S$.

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 [8] and compilers from the CADP toolbox [15] for obtaining LTSs from LNT specifications. LNT is an extension of LOTOS, an ISO-standardised process algebra [20], which supports data types, functions, and processes. LNT processes (see Figures 4 (a) and 5 (a)) are built from actions, choices (**select**), parallel composition (**par**), looping behaviours (**loop**), and sequential composition (**;**). For brevity, in this paper, we utilise only a fragment of the LNT syntax depicted in Table 1 to explain our approach. B stands for a LNT term, A for an action, E for a Boolean expression, x for a variable, and T for a type. LNT is formally defined using operational semantics based on LTSs [8]. The approach presented in the rest of this section is generic in the sense that it applies to deterministic LTSs produced from any compiler/verification tool and computed using classic determinisation techniques [19].

$$
\begin{array}{rcl}
B & ::= & A \mid B_1; B_2 \mid \textbf{select } B_1 [] ... [] B_n \textbf{ end select} \\
 & & \mid \textbf{par } B_1 || ... || B_n \textbf{ end par} \mid \textbf{while } E \textbf{ loop } B \textbf{ end loop} \\
 & & \mid \textbf{var } x{:}T \textbf{ in } B \textbf{ end var}
\end{array}
$$

Table 1: Fragment of LNT Syntax

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

Definition 2.2. (Trace) Given an LTS $M = (S, s^0, \Sigma, T)$, a trace τ of size $n \in \mathbb{N}$ is a sequence of actions $l_1, l_2, \ldots, l_n \in \Sigma$ 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 $t(M)$.

As labels may correspond to different transitions in an LTS, we introduce a slightly different notion called *path*, as a sequence of transitions of the LTS.

Definition 2.3. (Path) Given an LTS $M = (S, s^0, \Sigma, T)$, a path of size $n \in \mathbb{N}$ is a sequence of transitions $s^0 \xrightarrow{l_1} s_1 \in T, s_1 \xrightarrow{l_2} s_2 \in T, \ldots, s_{n-1} \xrightarrow{l_n} s_n \in T$. The set of all paths of M is written $p(M)$.

Model checking consists of verifying that an LTS model satisfies a given temporal property φ, which specifies some expected requirements of the system. Temporal properties are usually divided into two main families: safety and liveness properties [2]. In this work, we focus on liveness properties, which are widely used in the verification of real-world systems. Liveness properties state that "*something good eventually happens*". In particular, we focus on a class of liveness properties called *single inevitable execution* properties[1] which state that, given an LTS $M = (S, s^0, \Sigma, T)$ and an action $\alpha \in \Sigma$, every trace in $t(M)$ contains at least one action labelled α. This can be expressed as an LTL (Linear Temporal Logic [31]) formula $\Diamond\alpha$ (i.e., α is eventually executed). The action α is called the inevitable action.

An inevitable execution property can be semantically characterised by a possibly infinite set of traces $t_\varphi \subseteq t(M)$, corresponding to the traces that comply with the property φ in an LTS (i.e., correct traces). If the LTS model does not satisfy the property, the model checker returns a *counterexample*.

Definition 2.4. (Counterexample) Given an LTS $M = (S, s^0, \Sigma, T)$ and a property φ, a counterexample is any trace which belongs to $t(M) \setminus t_\varphi$.

The new contributions of this paper rely on existing results presented in [3, 4]. In this previous work, the 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.

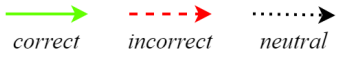


Figure 1: Types of transitions

In order to detect these faulty states, the transitions in the model first need to be classified into different types. The type of a transition indicates whether that transition belongs to paths satisfying the property or not. More precisely, transitions in the LTS are classified into three types: correct transitions satisfying the property, incorrect transitions violating the property, and neutral transitions not satisfying nor violating the property. These transition types are represented using different styles (and colours) as illustrated in Figure 1.

Definition 2.5. (Types of Transition) Given an LTS $M = (S, s^0, \Sigma, T)$ and a property φ, the transitions in T can be classified into three different types:

- A transition $t = s \xrightarrow{l} s' \in T$ is correct if $\forall p \in \{p' \in p(M) \mid t \in p'\}$, $p \models \varphi$. The set of correct transitions is written T_c.
- A transition $t = s \xrightarrow{l} s' \in T$ is incorrect if $\forall p \in \{p' \in p(M) \mid t \in p'\}$, $p \not\models \varphi$. The set of incorrect transitions is written T_i.
- A transition $t = s \xrightarrow{l} s' \in T$ is neutral if $\exists p, p' \in \{p'' \in p(M) \mid t \in p''\}$ such that $p \models \varphi$ and $p' \not\models \varphi$. The set of neutral transitions is written T_n.

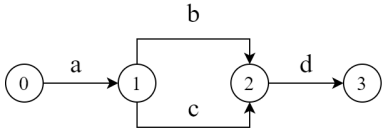
[1] A property with this type only contains a single action, in contrast, *nested inevitable execution* properties involve multiple actions.

The information concerning the transition type 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 $\gamma \in \Gamma$. An LTS in which each transition has been tagged with a type is called *tagged LTS*.

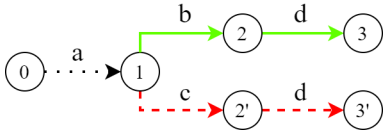
Definition 2.6. (Tagged LTS) Given an LTS $M = (S, s^0, \Sigma, T)$, and the set of transition tags Γ, the tagged LTS is a tuple $M_\Gamma = (S_\Gamma, s^0_\Gamma, \Sigma_\Gamma, T_\Gamma)$ where $S_\Gamma = S$, $s^0_\Gamma = s^0$, $\Sigma_\Gamma = \Sigma$, and $T_\Gamma \subseteq S_\Gamma \times \Sigma_\Gamma \times \Gamma \times S_\Gamma$.

In practice, tagging an LTS is often preceded by another process called *unfolding* [14, 28]. This process generates a new LTS in which no neutral transition is preceded by both incorrect and correct transitions, by duplicating the original problematic neutral transition and tagging it as either correct or incorrect. The resulting LTS contains more states and transitions than the original one, but is semantically equivalent to it.

In the rest of this section, the following dot operators are used to obtain the elements of a transition $t = s \xrightarrow{(l,\gamma)} s' \in T_\Gamma$ in the tagged LTS: $t.src = s$ and $t.tgt = s'$ return the source and target states, $t.l = l$ returns the label, $t.\gamma = \gamma$ returns the tag. Also, function $\Phi : 2^{T_\Gamma} \rightarrow 2^{\Sigma}$ retrieves the labels of a given set of transitions.



(a) Original LTS



(b) Unfolded & Tagged LTS

Figure 2: Tagging of LTS

Example. Figure 2 (a) shows an untagged LTS, while Figure 2 (b) depicts the tagged version of that LTS according to a property $\varphi = \Diamond b$. There are two paths in the untagged LTS: $p_1 = 0 \xrightarrow{a} 1, 1 \xrightarrow{b} 2, 2 \xrightarrow{d} 3$ and $p_2 = 0 \xrightarrow{a} 1, 1 \xrightarrow{c} 2, 2 \xrightarrow{d} 3$. Here, $p_1 \models \varphi$ and $p_2 \not\models \varphi$ because there is a transition labelled with b in p_1 while there is no such transition in p_2. In the tagged LTS, transitions $1 \xrightarrow{(b,correct)} 2$ and $1 \xrightarrow{(c,incorrect)} 2'$ are tagged as correct and incorrect because they only belong to, respectively, p_1 and p_2, whereas $0 \xrightarrow{(a,neutral)} 1$ is neutral because it belongs to both paths. Finally, $2 \xrightarrow{d} 3$ is unfolded into $2 \xrightarrow{(d,correct)} 3$ and $2' \xrightarrow{(d,incorrect)} 3'$ because it is preceded by both a correct and an incorrect transition.

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). In the specific case in which all the transitions of the tagged LTS are incorrect, the initial state is considered as a faulty state.

Definition 2.7. (Faulty State) Given a tagged LTS $M_\Gamma = (S_\Gamma, s^0_\Gamma, \Sigma_\Gamma, T_\Gamma)$ and the set of incorrect transitions $T_i \subseteq T_\Gamma$, a state $s \in S_\Gamma$ is a faulty state if and only if $((\forall t = s' \xrightarrow{(l,\gamma)} s \in T_\Gamma, \gamma = neutral) \wedge (\exists t' = s \xrightarrow{(l'',\gamma'')} s'', t'' = s \xrightarrow{(l''',\gamma''')} s''' \in T_\Gamma$ such that $\gamma'' \neq \gamma'''))) \vee (s = s^0 \wedge T_i = T_\Gamma)$.

By looking at the outgoing transitions of a faulty state, five kinds of faulty states can be identified:

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

These faulty states are represented using different styles of border (and colours) as depicted in Figure 3.

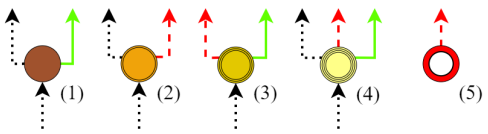


Figure 3: Faulty states

Finally, the notion of Counterexample LTS (CLTS) is defined and will be used in the rest of this paper.

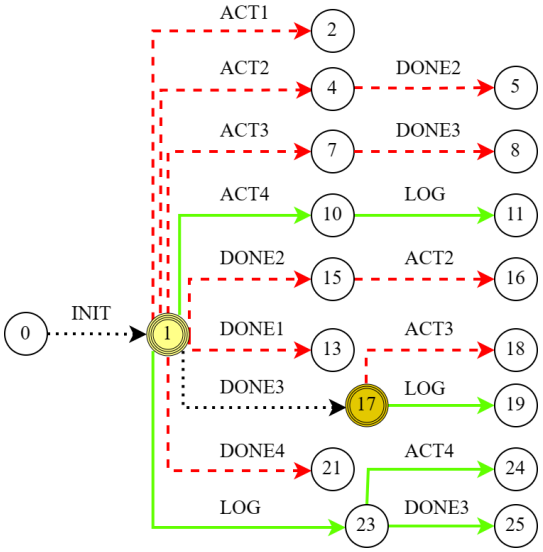
Definition 2.8. (Counterexample LTS) Given a tagged LTS $M_\Gamma = (S_\Gamma, s^0_\Gamma, \Sigma_\Gamma, T_\Gamma)$ and the set of faulty states F computed on this tagged LTS, the corresponding CLTS is the tuple $C = (M_\Gamma, F)$.

The reader interested in more details regarding the algorithms for computing tagged LTSs and CLTSs can refer to [3, 4]. Also, the use of CLTSs for debugging programs that violate safety properties has been proposed in [12, 13].

Example. Two examples of LNT specifications with their corresponding CLTSs are presented in Figures 4 and 5 (called examples A and B, respectively). Note that choices (i.e., **select**) in the LNT are represented in the CLTS as multiple transitions outgoing from the same state, whereas parallel compositions (i.e., **par**) are represented as interleaved executions [27]. Suppose that the property to be respected by both specifications states that action LOG eventually happens, which is written in LTL as $\Diamond$LOG. The specification in Figure 4 (a) contains a select statement with several choices (lines 4 to 16). LOG is not executed in some of these choices; hence, the CLTS in Figure 4 (b) contains faulty states and incorrect transitions. There is a loop statement (lines 7 to 10) inside a parallel composition

```
1  process main [INIT, ACT1, ACT2, ACT3, ACT4,
2  DONE1, DONE2, DONE3, DONE4, LOG: any] is
3    INIT ;
4    select
5      select DONE1 [] ACT1 end select
6    []
7      par DONE2 || ACT2 end par
8    []
9      par DONE3
10     || select LOG [] ACT3 end select
11     end par
12   []
13     select DONE4
14     [] par LOG || ACT4 end par
15     end select
16   end select
17 end process
```

(a) LNT specification



(b) CLTS

Figure 4: Example A

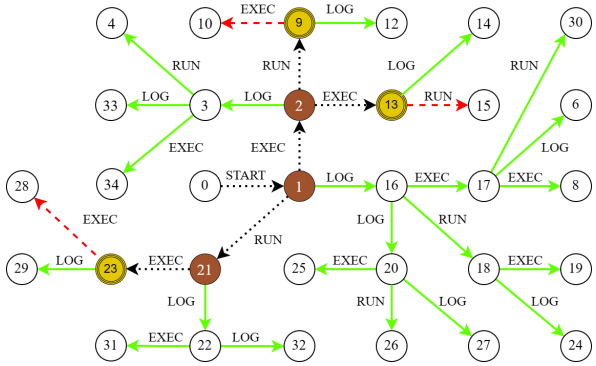
(lines 4 to 11) in the second specification (Figure 5 (a)). This causes some actions to appear more than once in some paths of the CLTS. These two running examples are used throughout the next section to illustrate the computation of patches.

3 AUTOMATED COMPUTATION OF PATCHES

A patch is used for correcting a specification that violates a single inevitable execution property. We first define the notion of patch, followed by several preliminary computations. Then, we detail the algorithm for computing the patches, along with a complexity analysis. Finally, we prove that every patch in the resulting set of patches is an *optimal* patch, that is, the patch of the smallest size having the smallest impact on the correct traces of the CLTS.

```
1  process main [START, RUN, EXEC, LOG: any] is
2    var x: Nat in
3      x := 0 ; START ;
4      par
5        select LOG [] RUN end select
6      ||
7        while (x < 2) loop
8          select LOG [] EXEC end select ;
9          x := x + 1
10       end loop
11     end par
12   end var
13 end process
```

(a) LNT specification



(b) CLTS

Figure 5: Example B

3.1 Patch

In this work, a patch is proposed to the user for modifying a specification that initially translates into a CLTS containing incorrect transitions to become another specification that translates into a CLTS composed of only correct transitions. This modification is made by adding the inevitable action α just before precise actions in the specification to make the execution of α become inevitable. Therefore, a patch is defined as a set of actions in the specification where the inevitable action should be added just before them.

Definition 3.1. (Patch) Given a CLTS $C = ((S_\Gamma, s^0_\Gamma, \Sigma_\Gamma, T_\Gamma), F)$ and an inevitable execution property $\varphi = \Diamond\alpha$, a patch p of size $n \in \mathbb{N}$ is a set of actions $\{l_1, l_2, ..., l_n\}$, $l_i \in \Sigma_\Gamma$. A patch is used to make the specification associated with C satisfy φ by adding α just before every action $l_i \in p$ in that specification.

It is worth noting that every action $l_i \in p$ except the inevitable action α, is assumed to appear only once in the specification. Consequently, the user can identify exactly where the action α should be added (i.e., if l_i appears multiple times, then it is not straightforward to identify before which of them α should be added). Nevertheless, the application of the approach is not limited to specifications with unique actions. If a specification contains actions that appear multiple times, then those actions can be temporarily renamed such that every action becomes unique.

Example. An example of a patch for the specification presented in Figure 4 (a) is p = {DONE1, ACT1, DONE2, ACT3, DONE4}. This

patch is obtained by ensuring that LOG is executed in every block of code in the select statement from lines 4 to 16. The first block (line 5) contains a select statement which executes either DONE1 or ACT1; these two actions must be included in the patch to make LOG inevitable. On the other hand, we may choose to include either DONE2 or ACT2 in the second block (line 7) because they are in a parallel statement. For the third block (lines 9 to 11), ACT3 is included in the patch instead of DONE3 because adding LOG before DONE3 would make it possible for LOG to be executed twice (i.e., impacting the correct part). There is a select statement in the last block (lines 13 to 15). In this case, only DONE4 must be added to the patch because there is already a parallel composition with a LOG action in the second block of this select statement.

This example shows that it is not straightforward to manually obtain a patch. A specification may contain hundreds of lines with nested select statements and parallel compositions that make it impossible to manually find a patch. One may propose to add the action LOG at the beginning or at the end; however, this would impact the correct parts of the specification. This paper presents a method to automatically compute a set of patches that is optimal in terms of size and (if any) in terms of impact on the correct parts of the specification.

3.2 Preliminary Computations

This section introduces the necessary information that should be retrieved from the CLTS before the computation of patches. First of all, we focus on the parts of the model where the property is not satisfied. These parts correspond to transitions tagged as incorrect. To understand the source of the violation, we need to identify the set of transitions associated with the precise starting point of the property's unsatisfaction. These are the incorrect transitions outgoing from the faulty states. We call them the *first incorrect transitions*.

Definition 3.2. (First Incorrect Transitions) The set of all first incorrect transitions in a CLTS $C = ((S_\Gamma, s^0_\Gamma, \Sigma_\Gamma, T_\Gamma), F)$ is defined as $T_{fit} \subseteq T_i \subseteq T_\Gamma$ where $\forall t \in T_{fit}, t.src \in F$.

As these transitions correspond to the starting point of property's violation, patching all of them necessarily makes the specification satisfy the property. Nonetheless, there may exist transitions in the CLTS that have the same labels but different tags, i.e., $t, t' \in T_\Gamma, t \neq t'$ such that $t.l = t'.l \wedge t.\gamma \neq t'.\gamma$. Consequently, there may exist transitions in T_{fit} that have the same labels as other transitions tagged as correct. It is important to avoid patching the actions associated to these transitions when possible because it would *impact the correct traces* of the CLTS.

Definition 3.3. (Impact on Correct Traces) Given a CLTS $C = ((S_\Gamma, s^0_\Gamma, \Sigma_\Gamma, T_\Gamma), F)$, the set of correct traces $t_c \subseteq t(C)$, a patch p, and a CLTS $C' = ((S'_\Gamma, s'^0_\Gamma, \Sigma'_\Gamma, T'_\Gamma), F')$ generated from the patched specification, t_c is said to be impacted by p if $\exists \tau \in t_c$ such that $\tau \notin t(C')$. This impact is quantified as the number of actions belonging to correct traces of C that no longer exist in C', i.e., $|\bigcup_{\tau=(l_0,\ldots,l_n)\in t_c,\ \tau\notin t(C')} \ \bigcup_{i\in[0\ldots n]} l_i|$.

Example. Suppose that for the specification in Figure 4 (a), a patch p = {INIT} is proposed. This patch would impact the correct parts of the specification because the correct traces in the initial CLTS, such as INIT, ACT4, LOG (see Figure 4 (b)), would no longer exist in the new CLTS (i.e., replaced by LOG, INIT, ACT4, LOG). In this specific case, impacting a correct trace means executing action LOG more than once.

To avoid impacting the correct traces of the CLTS, we must consider transitions in T_{fit} that would exclusively impact the incorrect traces of the CLTS once patched. These transitions are the transitions outgoing from faulty states and labelled with actions belonging to incorrect transitions only. This set of transitions is defined as the *set of exclusively incorrect transitions*.

Definition 3.4. (Exclusively Incorrect Transitions) For a given CLTS $C = ((S_\Gamma, s^0_\Gamma, \Sigma_\Gamma, T_\Gamma), F)$, the set of exclusively incorrect transitions is defined as $T_{eit} \subseteq T_{fit} \subseteq T_i \subseteq T_\Gamma$ where $\forall t \in T_{eit}, \nexists t' \in T_\Gamma, t' \neq t$ such that $t.l = t'.l \wedge t'.\gamma \neq incorrect$.

Ex.	Incorrect transitions (T_i)	First incorrect transitions (T_{fit})	Exclusively incorrect transitions (T_{eit})
A	(1, ACT1, 2) (1, ACT2, 4) (1, ACT3, 7) (1, DONE2, 15) (1, DONE1, 13) (1, DONE4, 21) (4, DONE2, 5) (7, DONE3, 8) (15, ACT2, 16) (17, ACT3, 18)	(1, ACT1, 2) (1, ACT2, 4) (1, ACT3, 7) (1, DONE2, 15) (1, DONE1, 13) (1, DONE4, 21) (17, ACT3, 18)	(1, ACT1, 2) (1, ACT2, 4) (1, ACT3, 7) (1, DONE2, 15) (1, DONE1, 13) (1, DONE4, 21) (17, ACT3, 18)
B	(9, EXEC, 10) (13, RUN, 15) (23, EXEC, 28)	(9, EXEC, 10) (13, RUN, 15) (23, EXEC, 28)	∅

Table 2: Sets of incorrect transitions

Example. Table 2 shows the computed sets of transitions T_i, T_{fit}, and T_{eit} from the CLTS in Figures 4 (b) and 5 (b). In example A, $T_{fit} \subset T_i$ because not every incorrect transition is outgoing from a faulty state (e.g., (15, ACT2, 16)). In example B, $T_i = T_{fit}$ because all three incorrect transitions are outgoing from faulty states. These transitions are not in T_{eit} since there are other transitions that are not incorrect transitions yet have the same labels (e.g., the correct transition (3, EXEC, 34) has the same label as (9, EXEC, 10)).

3.3 Computation of Patches

In this work, we propose to correct a specification by adding the inevitable action of the property before one or several actions of the specification. By doing so, the action of the property becomes inevitable in any execution of the specification (i.e., the property becomes satisfied). As explained in Section 3.1, the set of actions to be preceded by the inevitable one is called a patch. Interestingly, there may be several possible solutions for patching the model. In this paper, the proposed approach computes the *optimal* patches, that are, the patches of smallest size having (if any) the smallest impact on correct traces of the CLTS.

In this work, the reachability between transitions of the CLTS is used to facilitate the computation of patches. One transition is said to be reachable from another transition if there exists a sequence of transitions connecting them. Reachability can be useful for computing the patches. As an example, in Figure 4 (a), adding action LOG just before action INIT results in a CLTS composed of only correct transitions. This is because every incorrect transition in the CLTS is reachable from the transition labelled with INIT.

Definition 3.5. (Reachability of Transitions) Let $C = ((S_\Gamma, s^0_\Gamma, \Sigma_\Gamma, T_\Gamma), F)$ be a CLTS. $\forall t, t' \in T_\Gamma$, t' is said to be reachable from t if and only if $\exists s_1, s_2, ..., s_n \in S$ such that $t.tgt \xrightarrow{(l_1,\gamma_1)} s_1 \in T_\Gamma, s_1 \xrightarrow{(l_2,\gamma_2)} s_2 \in T_\Gamma$, ..., $s_n \xrightarrow{(l_{n+1},\gamma_{n+1})} t'.src \in T_\Gamma$.

A patch is more interesting than other patches when associated with a transition that can reach more incorrect transitions. In the following, we refer to the set of transitions that are reachable from a transition as the *reachable transitions set* of this transition.

Definition 3.6. (Reachable Transitions Set) Let $C = ((S_\Gamma, s^0_\Gamma, \Sigma_\Gamma, T_\Gamma), F)$ be a CLTS. $\forall t \in T_\Gamma$, we define the set of reachable transitions of t as $\delta(t) = \{t' \in T_\Gamma \mid t'$ is reachable from t$\} \cup \{t\}$.

By extension, we define $\delta_I(t)$ and $\delta_C(t)$ as the sets of reachable incorrect and correct transitions of t, respectively.

We have seen previously that patching an action of the specification was correcting all the transitions of the LTS that are labelled with this action. Thus, we define the *extended reachable transitions set* of a label l as the set of transitions reachable from all the transitions labelled with l.

Definition 3.7. (Extended Reachable Transitions Set) Let $C = ((S_\Gamma, s^0_\Gamma, \Sigma_\Gamma, T_\Gamma), F)$ be a CLTS. $\forall l \in \Sigma_\Gamma$, we define the extended reachable transitions set of l as $\Delta(l) = \bigcup_{\substack{t \in T_\Gamma \\ t.l = l}} \delta(t)$.

By extension, we define $\Delta_I(l)$ and $\Delta_C(l)$ as the extended sets of reachable incorrect and correct transitions of l, respectively.

The computation of patches is performed by Algorithm 1. It takes as input the three sets of incorrect transitions T_i, T_{fit}, and T_{eit} and returns the set of computed patches P. At line 1, the algorithm first determines whether the specification can be patched without impacting the correct part of the specification. This is feasible if all first incorrect transitions are exclusively incorrect, or if patching all exclusively incorrect transitions is enough to correct the specification. If this is the case, function *PatchIncorrectOnly* is executed. This function iterates over the set of exclusively incorrect transitions until finding combinations of actions for which the corresponding transitions can reach all the incorrect transitions of the CLTS, that is, correcting entirely the specification. When such combinations (i.e., patches) are found, they are added to the set of optimal patches of the specification. Function *PatchIncorrectAndCorrect* is executed if the condition at line 1 is not satisfied. It generates all possible combinations of the set of first incorrect transitions, and selects the smallest ones having the smallest impact on the correct traces of the CLTS, while correcting entirely the specification.

Algorithm 1 Algorithm for Computing Patches

Inputs: T_i, T_{fit}, T_{eit} (Transition sets defined in Section 3.2)
Output: P (Set of computed patches)

1: **if** $(T_{fit} = T_{eit}) \vee (\bigcup_{\forall t \in T_{eit}} \Delta_I(t.l) = T_i)$ **then**
2: $P \leftarrow$ PatchIncorrectOnly(T_{eit}, T_i)
3: **else**
4: $P \leftarrow$ PatchIncorrectAndCorrect(T_{fit}, T_i)
5: **end if**
6: **return** P

Algorithm 2 PatchIncorrectOnly

Inputs: T_i, T_{eit} (Transition sets defined in Section 3.2)
Output: P (Set of computed patches)

1: $PF \leftarrow$ False; $PS \leftarrow 1$; $P \leftarrow \emptyset$
2: **while** PF = False $\wedge$ $PS \leq |\Phi(T_{eit})|$ **do** ▷ iter. until patch found
3: $PP \leftarrow$ findAllCombinations(PS, $\Phi(T_{eit})$)
4: **for all** $pp \in PP$ **do**
5: **if** $\bigcup_{\forall l \in pp} \Delta_I(l) = T_i$ **then** ▷ pp corrects the spec
6: $PF \leftarrow$ True
7: $P \leftarrow P \cup \{pp\}$
8: **end if**
9: **end for**
10: $PS \leftarrow PS + 1$
11: **end while**
12: **return** P

Function *PatchIncorrectOnly* (Algorithm 2) computes the optimal patches in terms of the number of actions in the patch. First, it computes all subsets of actions belonging to T_{eit} of size $PS = 1$ with the function *FindAllCombinations(PS, $\Phi(T_{eit})$)*. Then it checks whether all the transitions of T_i can be reached from the transitions labelled with the actions in the current subset (line 5). If this is the case, this subset is sufficient to make the transitions in T_i become correct. Therefore, the subset is added to the set of patches P, and PF is set to *True* to indicate that a patch was found. Then, the next subset is analysed. If no subset of size PS is found (i.e., PF = *False*) at the end of an iteration, PS is set to $PS + 1$ and a new iteration starts.

Function *PatchIncorrectAndCorrect* (Algorithm 3) computes the optimal patches in terms of impact on the correct traces of the CLTS. It iterates through all possible sets of actions belonging to transitions in T_{fit} (lines 2 to 4). Each set of actions is a possible patch denoted as *pp*. The algorithm checks if the transitions in T_i are all reachable from the transitions labelled by *pp* (line 5). If this is the case, this patch is compared to already computed ones, if any. If it has less impact on the correct traces of the CLTS than previous patches (line 7), the previous patches are discarded and the current one becomes the only patch (line 8). If it has the same impact as previous patches, it is added to the set of patches (line 11). If it has more impact, it is discarded.

Finally, we provide the worst-case time complexity of the algorithm. As the algorithm computes all combinations of T_{fit}, in the worst case for both functions *PatchIncorrectOnly* and *PatchIncorrectAndCorrect*, the resulting complexity is exponential in the size

Algorithm 3 PatchIncorrectAndCorrect

Inputs: T_i, T_{fit} (Transition sets defined in Section 3.2)
Output: P (Set of computed patches)

1: $PF \leftarrow$ False; $PS \leftarrow 1$; $P \leftarrow \emptyset$; $TI \leftarrow \infty$
2: **while** $PS \leq |\Phi(T_{fit})|$ **do** ▷ iterate over all possible patches
3: $PP \leftarrow$ findAllCombinations(PS, $\Phi(T_{fit})$)
4: **for all** $pp \in PP$ **do**
5: **if** $\bigcup_{\forall l \in pp} \Delta_I(l) = T_i$ **then** ▷ pp corrects the spec
6: $CAI \leftarrow |\Phi(\bigcup_{\forall l \in pp} \Delta_C(l))|$ ▷ impact on correct traces
7: **if** $CAI < TI$ **then** ▷ less impact than current $p \in P$
8: $P \leftarrow \{pp\}$
9: $TI \leftarrow CAI$
10: **else if** $CAI = TI$ **then** ▷ same impact as curr. $p \in P$
11: $P \leftarrow P \cup \{pp\}$
12: **end if**
13: **end if**
14: **end for**
15: $PS \leftarrow PS + 1$
16: **end while**
17: **return** P

of T_{fit}: $O(|T_{fit}|^{\lceil \frac{|T_{fit}|}{2} \rceil + 1})$. However, experiments showed that both the size of T_{fit} and of the patches are generally small, which makes our approach execute efficiently on many realistic specifications in practice.

Example. The computation of patches for examples A and B is illustrated by presenting the list of iterations in Tables 3 and 4. The columns of the first table contain the following values (from left to right): the index of the iteration which also represents the size of the current set of actions, the current set of actions, the set of actions reachable from actions belonging to the current set of actions, the set of actions associated to the incorrect transitions, and boolean values representing the satisfaction of the property. In the second table, there are two additional columns: the set of actions of the correct transitions reachable from transitions labelled by the actions in the current set of actions (fourth column), and the size of that set (last column). These two additional columns correspond to the impact on the correct traces of the CLTS.

As shown in the first iteration of Table 3, patching less than five actions is not sufficient because it only impacts a subset of the actions associated with the set of incorrect transitions. Thus, in the case of example A, the algorithm returns $P = \{p_1 = \{\text{ACT1, ACT3, DONE1, DONE2, DONE4}\}, p_2 = \{\text{ACT1, ACT2, ACT3, DONE1, DONE4}\}\}$. Meanwhile in Table 4, any combination of actions in {RUN, EXEC} is sufficient to correct the specification. However, to be optimal in terms of impact on the correct traces of the CLTS, the algorithm returns the patches having the smallest number of impacted correct actions (last column). As a result, the optimal solution is to either patch only RUN or EXEC, i.e., $P = \{p_1 = \{RUN\}, p_2 = \{EXEC\}\}$.

The LNT specifications of both examples that have been corrected with the proposed patches are presented in Figure 6. As shown in both specifications, the LOG actions (highlighted with underlines) are added just before the actions in each patch. For instance in the one depicted in Figure 6 (b) at line 5, LOG is added just before RUN. In this case, the patch $p_1 = \{RUN\} \in P$ is used.

Itr.	Current set of actions	Reachable incorrect transition labels	Incorrect transition labels	Property satisfied
1	{ACT1}	{ACT1}	{ACT1, ACT2, ACT3, DONE1, DONE2, DONE3, DONE4}	*false*
	{ACT2}	{ACT2, DONE2}	{ACT1, ACT2, ACT3, DONE1, DONE2, DONE3, DONE4}	*false*
	...	...	...	...
5	...	...	...	...
	{ACT1, ACT3, DONE1, DONE2, DONE4}	{ACT1, ACT2, ACT3, DONE1, DONE2, DONE3, DONE4}	{ACT1, ACT2, ACT3, DONE1, DONE2, DONE3, DONE4}	*true*
	{ACT1, ACT2, ACT3, DONE1, DONE2, DONE4}	{ACT1, ACT2, ACT3, DONE1, DONE2, DONE3, DONE4}	{ACT1, ACT2, ACT3, DONE1, DONE2, DONE3, DONE4}	*true*

Table 3: Iterations to find patches for example A

It.	Current set of actions	Reachable incorrect transition labels	Reachable correct transition labels	Incor. trans. labels	Prop. satis.	Impact. correct actions
1	{RUN}	{RUN, EXEC}	{RUN}	{RUN, EXEC}	*true*	1
	{EXEC}	{RUN, EXEC}	{EXEC}	{RUN, EXEC}	*true*	1
2	{RUN, EXEC}	{RUN, EXEC}	{RUN, EXEC}	{RUN, EXEC}	*true*	2

Table 4: Iterations to find patches for example B

3.4 Optimality

This section first proves that every patch in the set of computed patches systematically makes the specification satisfy the property Then, it shows that every patch in the set of patches is optimal in terms of size when function *PatchIncorrectOnly* is called, and in terms of impact on the correct traces of the CLTS in both cases.

In the following propositions, we denote the set of computed patches as $P_C \subseteq 2^{\Sigma_\Gamma}$, and the set of all possible patches of the specification as $P_A \subseteq 2^{\Sigma_\Gamma}$. Also for brevity, we denote the number of

```
process main [INIT, ACT1, ACT2, ACT3, ACT4,
DONE1, DONE2, DONE3, DONE4, LOG: any] is
  INIT ;
  select
    select LOG ; DONE1 [] LOG ; ACT1 end select
  []
    par LOG ; DONE2 || ACT2 end par
  []
    par DONE3
    || select LOG [] LOG ; ACT3 end select
    end par
  []
    select LOG ; DONE4
    [] par LOG || ACT4 end par
    end select
  end select
end process
```

(a) LNT specification of example A

```
process main [START, RUN, EXEC, LOG: any] is
  var x: Nat in
    x := 0 ; START ;
    par
      select LOG [] LOG ; RUN end select
    ||
      while (x < 2) loop
        select LOG [] EXEC end select ;
        x := x + 1
      end loop
    end par
  end var
end process
```

(b) LNT specification of example B

Figure 6: Examples A and B after patching

impacted correct actions for a given patch, i.e., $|\Phi(\bigcup_{\forall l \in p} \Delta_C(l))|$, as a function $impact : 2^{\Sigma_\Gamma} \rightarrow \mathbb{N}$.

Proposition 1 states that for every patch belonging to the set of patches, the transitions labelled with the actions of the patch can impact all incorrect transitions in the CLTS. This implies that every patch can make all incorrect transitions become correct.

Proposition 1. *(Patch correctness).*
$\forall p \in P_C, \bigcup_{\forall l \in p} \Delta_I(l) = T_i$

Proof. The proof here is straightforward. A possible patch p is added to the set of patches if and only if the transitions labelled with its actions can reach all the incorrect transitions of the specification, i.e., if $\bigcup_{\forall l \in p} \Delta_I(l) = T_i$ (line 5 of Algorithms 2 & 3). □

Proposition 2 states that every patch in the set of patches has the same number of actions, and there exists no other patch with fewer actions that can make every incorrect transition become correct.

Proposition 2. *(Patch optimality in terms of size).*
$\forall p \in P_C, \nexists p' \in P_A \; s.t. \; (|p'| < |p| \; \wedge \bigcup_{\forall l \in p'} \Delta_I(l) = T_i)$

Proof (sketch). To prove this property, let us prove that Algorithm 2 is totally correct and returns the patches of smallest size. Let $T_\Phi = \Phi(T_{eit})$ and let assume that the built-in function *FindAllCombinations* : $\mathbb{N} \times \{T_\Phi\} \rightarrow 2^{T_\Phi}$ is sound and complete. The variant of the loop, which corresponds to the parameter of the *FindAllCombinations* function, is called *PS* and is initialised to 1. At the first iteration, all the combinations of size 1 of T_Φ are computed. The formula of Proposition 1 is then used to verify for each combination if it is a valid patch or not. If yes, the patch is added to the set of patches and the boolean value *PF* is set to *True* in order to break the loop at the next iteration. At the end of the first iteration, *PS* is incremented by one and the condition of the loop is reevaluated: either a patch was found (i.e., *PF* was set to true) or *PS* grew bigger than the size of T_Φ and the loop breaks, or a new loop starts with all the combinations of size 2, and so on. Thus, the loop necessarily terminates and returns a patch of minimal size (or no patch if the specification can not be corrected). □

Proposition 3 states that every patch in the set of patches has the same impact on the already correct traces of the CLTS, and that there exists no other patch with less impact that can make every single incorrect transition become correct.

Proposition 3. *(Patch optimality in terms of impact).*
$\forall p \in P_C, \nexists p' \in P_A \; s.t. \; (impact(p') < impact(p) \; \wedge \bigcup_{\forall l \in p'} \Delta_I(l) = T_i)$

Proof (sketch). To prove this property, let us prove that Algorithm 2 is totally correct and returns the patches of smallest impact. The proof of total correctness is identical to the one in Proposition 2. Let $T_\Phi = \Phi(T_{fit})$ and let assume that the built-in function *FindAllCombinations* : $\mathbb{N} \times \{T_\Phi\} \rightarrow 2^{T_\Phi}$ is sound and complete. The variant of the loop, which corresponds to the parameter of the *FindAllCombinations* function, is called *PS* and is initialised to 1. At the first iteration, all the combinations of size 1 of T_Φ are computed. The formula of Proposition 1 is then used to verify for each combination if it is a valid patch or not. If yes, the impact of the patch is computed using Definition 3.3. If this patch has a smaller impact than previous patches, the previous patches are discarded and the current patch is stored. If this patch has the same impact than previous patches, it is added to the set of patches. If this patch has a higher impact than previous patches, it is discarded. As all the possible patches are computed, the returned ones are by construction optimal. □

3.5 Applicability of Patches

The patches proposed in this approach offer the possibility to the user to make the specification correct with regards to the eventually property that was specified. However, as applying this patch modifies the specification, the new specification may violate some other properties that were previously satisfied. Thus, the user can also give as input to this approach a list of properties that should remain satisfied after the application of the patch. Once optimal patches have been computed, the user chooses one of these patches, and the corresponding patched specification is model checked using CADP [15]. If the patched specification no longer satisfies the properties given by the user, the approach returns a warning list with all the violated properties for this patch. The user can then

Specification	Spec. Size (loc)	CLTS Size (States / Transitions)	Size of T_{fit}	Size of the Patch	Alg.	CLTS Generation Time	Patches Computation Time	Global Execution Time
1. Backward [13]	50	69/78	6	5	2	0.91ms	1.52ms	3.62ms
2. Iteration [5]	23	112/159	1	1	3	0.75ms	0.09ms	2.21ms
3. Muller [26]	43	204/518	3	1	3	3.11ms	2.18ms	7.47ms
4. Omprace [34]	21	306/808	1	1	2	4.49ms	0.52ms	8.95ms
5. Interleaving [5]	42	501/1k	1	1	2	10.2ms	0.67ms	16.2ms
6. IoT [23]	507	1k/4.3k	46	1	2	27.1ms	24.3ms	93.5ms
7. Ifttt [12]	215	1k/2.9k	1	1	3	26.6ms	0.90ms	35.3ms
8. Causality [5]	116	1.7k/5.4k	2	1	3	31.0ms	4.83ms	52.1ms
9. BRP [16]	329	3.4k/5.4k	7	1	3	34.8ms	26.1ms	90.8ms
10. Station [4]	97	3.8k/14k	1	1	2	216ms	3.25ms	261ms
11. Fail. Manag. [29]	1.9k	11k/21k	20	1	3	192ms	45.9m	45.9m

Table 5: Results of the performance study on realistic examples

decide if (s)he wants to apply another optimal patch. If none of the proposed patches is relevant for the user, (s)he can ask for computing all the non-optimal patches. Similarly, (s)he chooses one of the non-optimal patches, and the corresponding patched specification is generated and model checked using CADP. If none of the non-optimal patches is relevant for the user, then it means that at least two properties that must be verified by the specification are conflicting (i.e., both can not be satisfied at the same time). In such a case, the specification requires manual revision in order to make it compliant with these properties.

4 TOOL SUPPORT

The proposed approach has been fully implemented in Python and is available online [1]. Its execution flow is detailed in Figure 7.

Starting from an LTS and a single inevitable property written in MCL, the tool generates the CLTS containing all the counterexamples of the property. Then, it computes the three transition sets defined earlier, namely T_i, T_{fit}, and T_{eit}. Finally, it applies either function *PatchIncorrectOnly* or *PatchIncorrectAndCorrect*.

As these two functions, and particularly *PatchIncorrectAndCorrect*, perform an exhaustive exploration of the state-space, the user can specify a time bound that the approach should try not to exceed. Indeed, when the bound is reached, the approach stops when it founds a patch, that is, immediately if a patch had already been found. The tool finally returns the list of patches found to the user, in textual format. It is worth noting that these patches are necessarily optimal when no bound is specified, or if the time bound has not been reached. Finally, the user picks the one making the most sense for him. To present the result of the patching, a correct specification (patched with the first patch of the list) is also returned to the user. When function *PatchIncorrectAndCorrect* is called, a warning message indicating that applying the patch will necessarily impact the correct traces of the specification is also returned.

Table 5 shows a performance analysis consisting of several realistic examples. The table contains the following columns (from left to right): the name of the specification, the size of the specification (in number of lines of LNT), the size of the corresponding CLTS (in number of states and transitions), the size of the T_{fit} set which is the basis of our computation (in number of transitions), the size of the computed patches (in number of actions), the algorithm used to compute the patches (with or without impact on the correct traces), the time elapsed to compute the CLTS, the time elapsed to compute the patches, and the global execution time of the tool chain. This performance analysis has been conducted on an HP EliteBook x360 1030 G8 Notebook PC running with an Intel Core i5-1145G7 @ 2.60GHz VPRO and 16GB of RAM.

The first ten examples run in very short execution time (a few hundreds of milliseconds at most). This is because either correct parts of the specification are not impacted and the patches found are small (examples 1, 4, 5, 6 & 10), or because the number of candidate transitions (i.e., the size of T_{fit}) is small (examples 2, 3, 7, 8 & 9). The last example takes more time to execute. Indeed, in this case, we need to compute all the possible patches in order to minimise the impact. Since T_{fit} contains 20 transitions in this case, the computation of patches inevitably takes more time because all combinations of T_{fit} are computed. Overall, the scalability issues are mostly due to the size of T_{fit}, but can often be drastically reduced without much impact on the results by using the time bound. This is indeed the case for example 11, in which the total computation lasts 45m, but the optimal patch is in fact computed in less that 5s.

5 RELATED WORK

Controller synthesis [32, 33] focuses on the generation of controllers with respect to a given system (called plant) designed as a finite automaton and properties to be ensured. Runtime verification [18, 24] is an alternative to traditional formal verification techniques, such as model checking, and aims at verifying whether an execution trace satisfies a given correctness property. Runtime enforcement [22, 25] goes beyond classic runtime verification by correcting the execution that deviates from its expected behaviour to ensure the satisfaction of a given property. Both controller synthesis and runtime enforcement aim at influencing the behaviour of the program from an external point of view, whereas the goal of our approach is to correct the identified bug by changing the corresponding specification.

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. [7] detects a set of causes for the failure of the

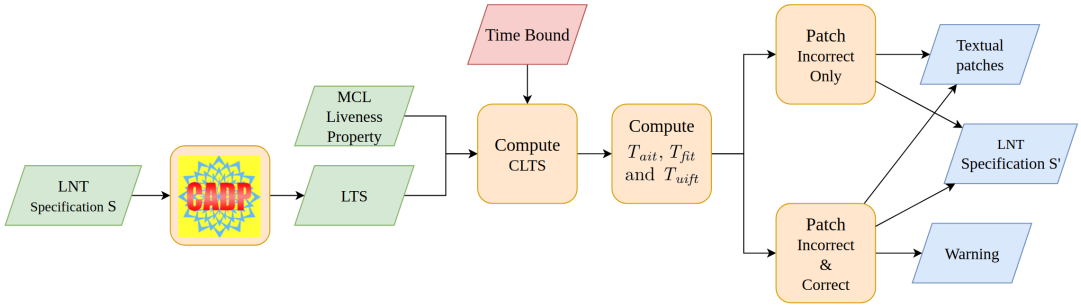


Figure 7: Overview of the Tool

specification from a given counterexample trace, marks them as red dots and presents the result to the user as a visual explanation of the failure. In [17], the authors present a general approach for causality analysis of system failures based on component specifications and observed component traces. In [6], 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 not only to detect bugs but also to propose corrections of the specification to avoid property violation.

In [21] the authors propose a method to interpret counterexamples from 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. Their 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 [21] aim at building an explanation from the counterexamples. Their approach produces an enhanced counterexample where inevitable events (w.r.t. the bug) are highlighted whereas we propose a patch to make the specification satisfy the property.

Fault localisation for program debugging has been an active topic of research for many years in the software engineering community [35]. One of the main approaches in that line of work aims at localising faults using testing approaches. As an example, the approach presented in [30] relies on mutation testing to locate effectively the faulty statements. Experiments carried out in [30] 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.

In [3], a method is presented for improving the comprehension of counterexamples returned by a model checker when an inevitability property is not satisfied on a given behavioural model. This approach first extends the model with prefix / suffix information w.r.t. the given property. This enriched model is then analysed to identify specific states consisting of a choice between transitions leading to a correct or incorrect part of the model. These specific states are exploited in order to propose a set of simplification techniques to extract relevant information that explains the cause of the bug. In this work, we go beyond building explanations when liveness properties are not satisfied since we also propose patches for the erroneous specification.

The authors in [13] propose techniques for counting the number of safety property violations in a program specification. The approach starts with generating the behavioural model of a given specification and a safety property. Afterwards, specific modifications are made to the specification. In every modification, the model is analysed to identify whether the part where the changes are made corresponds to a bug or not. In comparison with our work, this approach focuses only on quantifying the bugs, while ours is to find and patch them. Moreover, our method targets liveness properties, while their approach is aimed at safety properties.

6 CONCLUDING REMARKS

In this paper, we have presented some techniques for correcting automatically a specification of a concurrent program violating a given eventually property, which is a liveness property often used in practice by developers. To do so, we first use existing techniques in order to identify what parts of the model do not respect the property. Some analysis algorithms are then applied to more precisely traverse these erroneous parts in order to compute the minimal number of changes to be made on the specification to satisfy the property. The approach is fully automated given a specification and an eventually property as input. If there are several eventually properties that are violated, we apply the same process for each unsatisfied property. We evaluated the proposed techniques on several realistic examples, and performance results are satisfactory.

Although, in this initial work, we have focused on *simple* eventually properties, it is worth noting that optimally patching a specification with respect to such a property may be very difficult to achieve manually on realistic specifications. The approach proposed in this paper helps the user in this task by providing a fully automated solution. The main perspective of this work is to consider a larger family of liveness properties. Concretely, we plan to consider *nested* eventually properties, but this will oblige us to entirely adapt the core of our approach, which currently does not work for such properties.

ACKNOWLEDGMENTS

This work is supported by the French National Research Agency in the framework of the « France 2030 » program (ANR-15-IDEX-0002) and by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01).

REFERENCES

[1] 2023. Eventually Properties Patching Tool. https://anonymous.4open.science/r/EventuallyPropertiesPatcher-0334.

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

[3] Gianluca Barbon, Vincent Leroy, and Gwen Salaün. 2018. Counterexample Simplification for Liveness Property Violation. In *Proc. of SEFM'18 (LNCS, Vol. 10886)*. Springer, 173–188. https://doi.org/10.1007/978-3-319-92970-5_11

[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. https://doi.org/10.1109/TSE.2019.2915303

[5] Gianluca Barbon, Vincent Leroy, Gwen Salaün, and Emmanuel Yah. 2019. Visual Debugging of Behavioural Models. In *Proc. of ICSE'19*. IEEE / ACM, 107–110. https://doi.org/10.1109/ICSE-COMPANION.2019.00050

[6] Adrian Beer, Stephan Heidinger, Uwe Kühne, Florian Leitner-Fischer, and Stefan Leue. 2015. Symbolic Causality Checking Using Bounded Model Checking. In *Proc. of SPIN'15 (LNCS, Vol. 9232)*. Springer, 203–221. https://doi.org/10.1007/978-3-319-23404-5_14

[7] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard J. Trefler. 2012. Explaining counterexamples using causality. *Formal Methods Syst. Des.* 40, 1 (2012), 20–40. https://doi.org/10.1007/S10703-011-0132-2

[8] 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.

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

[10] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). 2018. *Handbook of Model Checking*. Springer. https://doi.org/10.1007/978-3-319-10575-8

[11] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. 1999. Patterns in Property Specifications for Finite-State Verification. In *Proc. of ICSE'99*. ACM, 411–420. https://doi.org/10.1145/302405.302672

[12] 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

[13] Irman Faqrizal and Gwen Salaün. 2022. Counting Bugs in Behavioural Models using Counterexample Analysis. In *Proc. of FormaliSE@ICSE'22*. ACM, 12–22. https://doi.org/10.1145/3524482.3527647

[14] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Proc. of ACM'05*. ACM, 110–121. https://doi.org/10.1145/1040305.1040315

[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. https://doi.org/10.1007/S10009-012-0244-Z

[16] Hubert Garavel, Julian Jacques Maurer, and Jose-Ignacio Requeno. 2015. Bounded Retransmission Protocol (CADP, demo 16). (2015).

[17] Gregor Gößler and Daniel Le Métayer. 2013. A General Trace-Based Framework of Logical Causality. In *Proc. of FACS'13 (LNCS, Vol. 8348)*. Springer, 157–173. https://doi.org/10.1007/978-3-319-07602-7_11

[18] Klaus Havelund and Allen Goldberg. 2005. Verify Your Runs. In *Proc. of VSTTE'05 (LNCS, Vol. 4171)*. Springer, 374–383. https://doi.org/10.1007/978-3-540-69149-5_40

[19] John E. Hopcroft and Jeffrey D. Ullman. 2000. *Introduction to Automata Theory, Languages and Computation, Second Edition*. Addison-Wesley.

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

[21] 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. https://doi.org/10.1007/3-540-46002-0_31

[22] Raphaël Khoury and Sylvain Hallé. 2015. Runtime Enforcement with Partial Control. In *Proc. of FPS'15 (LNCS, Vol. 9482)*. Springer, 102–116. https://doi.org/10.1007/978-3-319-30303-1_7

[23] Ajay Krishna, Michel Le Pallec, Radu Mateescu, and Gwen Salaün. 2022. Design and Deployment of Expressive and Correct Web of Things Applications. *ACM Trans. Internet Things* 3, 1 (2022), 1:1–1:30. https://doi.org/10.1145/3475964

[24] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *J. Log. Algebraic Methods Program.* 78, 5 (2009), 293–303. https://doi.org/10.1016/j.jlap.2008.08.004

[25] Jay Ligatti and Srikar Reddy. 2010. A Theory of Runtime Enforcement, with Results. In *Proc. of ESORICS'10 (LNCS, Vol. 6345)*. Springer, 87–100. https://doi.org/10.1007/978-3-642-15497-3_6

[26] Radu Mateescu, Wendelin Serwe, Aymane Bouzafour, and Marc Renaudin. 2020. Modeling an Asynchronous Circuit Dedicated to the Protection Against Physical Attacks. In *Proc. of MARS'20 (EPTCS, Vol. 316)*. 200–239. https://doi.org/10.4204/EPTCS.316.8

[27] Robin Milner. 1989. *Communication and concurrency*. Prentice Hall.

[28] M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. 1995. Transition-Systems, Event Structures, and Unfoldings. *Information and Computation* 118, 2 (1995), 191–207. https://doi.org/10.1006/inco.1995.1062

[29] Umar Ozeer, Gwen Salaün, Loïc Letondeur, François-Gaël Ottogalli, and Jean-Marc Vincent. 2020. Verification of a Failure Management Protocol for Stateful IoT Applications. In *Proc. of FMICS'20 (LNCS, Vol. 12327)*. Springer, 272–287. https://doi.org/10.1007/978-3-030-58298-2_12

[30] Mike Papadakis and Yves Le Traon. 2014. Effective Fault Localization via Mutation Analysis: A Selective Mutation Approach. In *Proc. of SAC'14*. ACM, 1293–1300. https://doi.org/10.1145/2554850.2554978

[31] Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proc. of FOCS'77*. IEEE, 46–67. https://doi.org/10.1109/SFCS.1977.32

[32] P.J.G. Ramadge and W.M. Wonham. 1989. The control of discrete event systems. *Proc. IEEE* 77, 1 (1989). https://doi.org/10.1109/5.21072

[33] P.J.G. Ramadge and W. M. Wonham. 1987. Supervisory Control of a Class of Discrete Event Processes. *SIAM Journal on Control and Optimization* 25, 1 (1987), 206–230. https://doi.org/10.1137/0325013

[34] Gwen Salaün and Lina Ye. 2015. Debugging Process Algebra Specifications. In *Proc. of VMCAI'15 (LNCS, Vol. 8931)*. Springer, 245–262.

[35] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368