Counterexample-Based Error Localization of Behavior Models

Tsutomu Kumazawa and Tetsuo Tamai

Graduate School of Arts and Sciences, The University of Tokyo,
Tokyo, Japan
{kumazawa,tamai}@graco.c.u-tokyo.ac.jp

Abstract. Behavior models are often used to describe behaviors of the system-to-be during requirements analysis or design phases. The correctness of the specified model can be formally verified by model checking techniques. Model checkers provide counterexamples if the model does not satisfy the given property. However, the tasks to analyze counterexamples and identify the model errors require manual labor because counterexamples do not directly indicate where and why the errors exist, and when liveness properties are checked, counterexamples have infinite trace length, which makes it harder to automate the analysis. In this paper, we propose a novel automated approach to find errors in a behavior model using an infinite counterexample. We find similar witnesses to the counterexample then compare them to elicit errors. Our approach reduces the problem to a single-source shortest path search problem on directed graphs and is applicable to liveness properties.

Keywords: Requirements Analysis, Design, Model Checking, Error Localization.

1 Introduction

Model Driven Engineering (MDE) is being accepted as a practical approach to develop reliable software efficiently [24]. Following MDE, a model of the software-to-be is built first, which goes through a series of model transformations to derive final code. It is obvious that the whole scheme crucially depends on correctness and appropriateness of the initial model.

As widely acknowledged, *model checking* [5] is one of the most powerful methods for formally validating correctness and appropriateness of a given model. The mostly used type of model checking technique takes behavior models represented as state machines as its target and checks if a given set of properties hold, employing graph searching algorithms or symbolic logical formula decision algorithms. Comparing the model checking approach to the theorem proving approach, one of the advantages of the former is often attributed to its capability of presenting counterexamples when verification fails. E. Clarke writes "It is impossible to overestimate the importance of the counterexample feature [4]."

But in practice, difficulties arise after counterexamples are obtained. Counterexamples do not directly indicate where in the model the errors that cause

M. Bobaru et al. (Eds.): NFM 2011, LNCS 6617, pp. 222–236, 2011.
© Springer-Verlag Berlin Heidelberg 2011

them exist. It is up to the developer's effort and intuition to find the part of the model that should be fixed to prevent occurrence of the counterexamples.

In this paper, we propose a new method and a tool that help developers fix errors in their models based on the detected counterexamples. In the research field of software model checking and debugging of source code, there exist a certain number of techniques that explain counterexamples and localize errors [1,13,3,12,11,16]. However, those existing methods for programs [1,13,12,11] only treat the violation of *safety properties* due to the limitation of software model checkers for source programs. Counterexamples for safety properties are by nature composed of event traces with finite length. Then it is relatively easy to automate identification of bad events in the trace. On the other hand, error localization for *liveness properties* poses a greater challenge, because in general it requires analysis of infinite-length counterexamples.

The state space of a program at the source code level is in general quite huge and when there is a loop structure, it is hard to decide when to stop expanding the state model graph. Techniques such as predicate abstraction are used to circumvent the problem. It is all right if a counterexample against some safety property is found in the current abstraction, because it can safely be concluded that the program violates the property. Otherwise, to explore the unsearched space, loops have to be expanded and it is not easy to decide when to stop the search. To deal with liveness properties, it induces much harder problems, because it is essential to identify precise loop structures (strongly connected components) and moreover even if counterexamples are found in an abstracted model, it is not sound to conclude that the original concrete model also violates the liveness property.

There are some pieces of work trying to treat liveness properties [3,16]. However, they have limitations such that it involves highly expensive computational complexity [3] or only specific kinds of liveness properties are supported [16].

In this paper, we propose LLL-S, a novel error localization technique in the given behavior model. We address the problem of analyzing an infinite trace, which takes the form of a finite prefix followed by an infinite cycle. Our idea is to find infinite and lasso-shaped witnesses (traces that satisfy the property) that resemble the given counterexample, and identify events to be modified by comparing each witness with the counterexample. We report all transitions that trigger the differences as candidate errors and the corresponding witnesses as their explanations. We use a Büchi automaton recognizing the target property as a set of witnesses, and adopts the edit distance between strings to measure distances between infinite and lasso-shaped traces. We find appropriate witnesses based on the distance by solving a single-source shortest path search problem on the Büchi automaton. LLL-S can be applied to the safety property class as well, where the length of counterexamples is finite and that of witness traces is infinite.

The main contributions of LLL-S are as follows. LLL-S can be applied to any Linear Temporal Logic formulas [9,21], including both liveness properties and safety properties. LLL-S focuses on errors in the behavior models that are used in MDE. We do not have to prepare a set of witnesses in advance, because it is

given as a Büchi automaton. Since LLL-S is based on well-established techniques combined together, it is easily automated.

Section 2 presents a motivating example. Section 3 explains the background. We explain LLL-S in Section 4. In Section 5, we report the results of the tool implementation of LLL-S and some case studies. We discuss some issues concerning our work and introduce related work in Section 6, and conclude in Section 7.

2 Motivating Example

Consider a concurrent system with a semaphore [21], CSys, whose LTS is shown in Fig. 1 (a). A LTS is a finite state machine described in terms of events. CSys consists of three processes: p.1, p.2 and Sema. The initial states of the processes are labeled 0. Two processes p.1 and p.2 repeatedly enter and leave the critical region by $p.\{1,2\}.enter$ and $p.\{1,2\}.exit$, respectively. Their exclusive access to the critical region is controlled by the mutual exclusion mechanism of the semaphore process Sema such that $p.i.mx.down$ $(i = 1, 2)$ lets p.i enter the critical region, and blocks the entrance of the other process until $p.i.mx.up$ occurs. The transitions sharing source and destination states are depicted by a single arrow. For example, the transition $(0,\ p.\{1,2\}.mx.up, 1)$ of Sema denotes $(0,\ p.1.mx.up, 1)$ and $(0,\ p.2.mx.up, 1)$. The behavior of CSys is presented by *parallel composition* [21] of three processes, which is based on interleaving of unshared events and simultaneous executions of shared events.

To verify the correctness of CSys's behavior, consider the *fluent Linear Temporal Logic* property $\mathsf{EXIT1} = \mathbf{G}(p.1.enter \Rightarrow \mathbf{F}p.1.exit)$, where $\mathbf{G}$, $\mathbf{F}$ and $\Rightarrow$ respectively denote *always*, *eventually* and *implication*. EXIT1 says that when p.1 enters the critical region, p.1 eventually leaves it. However, CSys does not satisfy EXIT1, because when p.1 stays in the critical region, p.2 can access it infinitely many times. The flaw in CSys is the incorrect mutual exclusion mechanism realized by Sema. A counterexample is $\pi^c = PC^\omega$, where the prefix P and the cycle C are finite event sequences shown in Table 1. The problem is to identify erroneous transitions of Sema.

We will find a witness $\tau = P'C'^\omega$ that are closest to π^c. A set of witnesses is given by a Büchi automaton recognizing EXIT1, $B(\mathsf{EXIT1})$ in Fig. 1 (b). Its initial state is b_0 and its event set is A_1, identical to the event set of CSys. Term (p, A, q) represents the transitions that share the source state p and destination

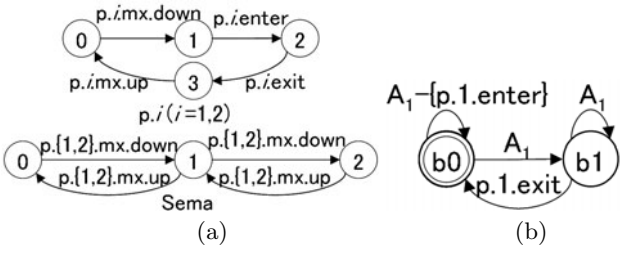

Fig. 1. Concurrent System CSys (a) and $B(\mathsf{EXIT1})$ (b)

Table 1. Counterexample of CSys for EXIT1 (π^c), and Witnesses (from τ^1 to τ^4)

π^c=[*p.1.mx.down*, *p.1.enter* (*p.2.mx.down*, *p.2.enter*, *p.2.exit*, *p.2.mx.up*)$^\omega$]
τ^1=[*p.1.mx.down*, *p.1.enter*, *p.1.exit* (*p.2.mx.down*, *p.2.enter*, *p.2.exit*, *p.2.mx.up*)$^\omega$]
τ^2=[*p.1.mx.down*, *p.1.enter* (*p.2.mx.down*, *p.1.exit*, *p.2.exit*, *p.2.mx.up*)$^\omega$]
τ^3=[*p.1.mx.down*, *p.1.enter*, *p.1.exit* (*p.2.mx.down*, *p.2.enter*, *p.2.exit*)$^\omega$]
τ^4=[(*p.1.mx.down*, *p.1.enter*, *p.1.exit*, *p.2.mx.down*, *p.2.enter*, *p.2.exit*, *p.2.mx.up*)$^\omega$]

state q, and whose events constitute the set A. To explain the advantage of using the closest witnesses to the counterexample for error localization and show the limitation of existing techniques, consider witnesses accepted by B(EXIT1) shown in Table 1. One of the simplest distances between π^c and τ is the number of edit operations required in transforming PC into $P'C'$ ignoring the cycling of C and C' (denoted by $d_e(\pi^c, \tau)$), whose concept is almost the same as those proposed by Chaki *et al.* [3] and Groce *et al.* [12].

The witness τ^1 is the closest to π^c because at least one insertion of *p.1.exit* after the second event *p.1.enter* of π^c should be applied to make π^c satisfy EXIT1 ($d_e(\pi^c, \tau^1) = 1$). The witness τ^1 tells us that p.1 should leave the critical region before p.2 enters there. Thus, τ^1 indicates that Sema does not correctly control p.1 and p.2's access to the critical region, and that the events other than *p.1.enter* have nothing to do with making π^c satisfy EXIT1. By comparing π^c with τ^1, we know the erroneous transitions (0, *p.1.mx.down*, 1) and (1, *p.2.mx.down*, 2) of Sema that are the nearest to the inserted event *p.1.enter*. These transitions show that Sema allows p.2 to enter the critical region by *p.2.mx.down* after it allows p.1 to enter there by *p.1.mx.down* but without following *p.1.mx.up*.

Another witness τ^2 is also the closest to π^c, i.e. $d_e(\pi^c, \tau^2) = 1$. Their difference is interpreted that *p.2.enter* of π^c should be forbidden. Therefore, p.2 should not enter the critical region infinitely many times when p.1 stays there, and the transition (1, *p.2.mx.down*, 2) of Sema enables p.2 to enter the region.

However, the witness τ^3 is unsuitable for showing the errors because τ^3 additionally requires the deletion of *p.2.mx.up* from τ^1, which is an unnecessary operation to make π^c satisfy EXIT1 (i.e. $d_e(\pi^c, \tau^3) = 2$). This deletion may mislead the developers into believing that *p.2.mx.up* should not occur. The distance d_e appropriately shows that τ^1 and τ^2 are closer to π^c than τ^3, and that τ^3 must not be used for error localization.

The witness τ^4 is the closest to π^c according to d_e because τ^1 and τ^4 consist of the same finite event sequence. However, τ^4 does not provide useful information to determine whether every event in P should be repeated infinitely many times, or P does not contain errors and *p.1.exit* is the only significant event to modify the violation of EXIT1 as the case of τ^1. Thus, we wish to judge that τ^4 is *not* as close to π^c as τ^1, but d_e does not work for our purpose. The cause of the problem is that d_e does not separate differences between the prefixes and the cycles of π^c and τ^4. Other existing methods [15,27,1,13,23,6,11] have the similar limitation due to their assumption that traces are of finite-length.

To summarize, it is desirable for error localization to obtain τ^1 and τ^2, but not τ^3 or τ^4 . In Section 4, we present a novel method to automatically find such witnesses based on a specific distance and the errors in CSys.

3 Background

A LTS is a tuple $L = (S, A, \Delta, s_0)$, where S is a finite set of states, A is a set of events, $\Delta \subseteq S \times A \times S$ is a transition relation, and $s_0 \in S$ is the initial state. A *trace* of L is a sequence of events $\pi = [a_0, a_1, \ldots, a_{n-1}]$ $(\forall 0 \leq i < n.(s_i, a_i, s_{i+1}) \in \Delta)$. For π, the sequence of states $[s_0, s_1, \ldots, s_n]$ is called a *path* of π. If $n = \infty$, we call π an infinite trace. Otherwise, we call π a finite trace. A set of all traces of L is denoted by $Tr(L)$. The suffix of a trace $\pi \in Tr(L)$ from a_i is denoted by $\pi[i]$. A transition $(s, a, s) \in \Delta$ is called a *self transition*.

Concerning model checking on L, a Büchi automaton-based technique has been proposed for FLTL [9]. A fluent is an atomic proposition whose truth value is determined over occurrence of events appearing in a trace. A fluent is a tuple $\mathsf{fl} = (I_{\mathsf{fl}}, T_{\mathsf{fl}}, b_{\mathsf{fl}})$, where $I_{\mathsf{fl}}, T_{\mathsf{fl}} \in A$ are a set of initiating and terminating events respectively such that $I_{\mathsf{fl}} \cap T_{\mathsf{fl}} = \emptyset$, and $b_{\mathsf{fl}} \in \{\mathbf{t}, \mathbf{f}\}$ is the initial truth value. For $\pi \in Tr(L)$, $\pi[i]$ satisfies fl $(\pi[i] \models \mathsf{fl})$ iff one of the following conditions holds: either $b_{\mathsf{fl}} \wedge (\forall j \in \mathcal{N}.0 \leq j \leq i \Rightarrow a_j \notin T_{\mathsf{fl}})$, or $\exists j \in \mathcal{N}.(j \leq i \wedge a_j \in I_{\mathsf{fl}}) \wedge (\forall k \in \mathcal{N}.j < k \leq i \Rightarrow a_k \notin T_{\mathsf{fl}})$. The set of fluents considered is denoted by FL.

A FLTL formula is defined inductively with the boolean and temporal operators as follows: $\phi, \psi = \mathbf{t} \mid \mathsf{fl} \in \mathsf{FL} \mid \phi \wedge \psi \mid \neg\phi \mid \mathbf{X}\phi \mid \phi\mathbf{U}\psi$. Given a trace $\pi \in Tr(L)$, the satisfaction operator $\models$ is defined inductively as follows:

$$\begin{array}{lll} \pi \models \mathbf{t}, & \pi \models \mathsf{fl} \in \mathsf{FL} \text{ iff } \pi[0] \models \mathsf{fl}, & \pi \models \neg\phi \text{ iff } \pi \not\models \phi, \\ \pi \models \phi \wedge \psi \text{ iff } (\pi \models \phi) \text{ and } (\pi \models \psi), & & \pi \models \mathbf{X}\phi \text{ iff } \pi[1] \models \phi, \\ \pi \models \phi\mathbf{U}\psi \text{ iff } \exists j \geq 0.\pi[j] \models \psi \text{ and } \forall 0 \leq i < j.\pi[i] \models \phi. & & \end{array}$$

Other operators are derived from the above operators: $\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$, $\phi \Rightarrow \psi = \neg\phi \vee \psi$, $\mathbf{F}\phi = \mathbf{t}\mathbf{U}\phi$ and $\mathbf{G}\phi = \neg\mathbf{F}\neg\phi$. We define $L \models \phi$ (L satisfies ϕ) iff $\forall \pi \in Tr(L).\pi \models \phi$. FLTL formulas are classified into safety and liveness properties [21]. A safety property such as $\mathbf{G}\neg p$ asserts that nothing bad ever happens, while a liveness property such as $\mathbf{F}p$ asserts that something good will eventually happen. EXIT1 is an instantiation of liveness properties.

Model checking on LTS L for FLTL formula ϕ is conducted as follows [9]: 1) build a Büchi automaton that accepts all traces satisfying $\neg\phi$, $B(\neg\phi)$, 2) build the parallel composition of L and $B(\neg\phi)$, and 3) search for an accepting trace, which is a *counterexample*. A Büchi automaton $B = (S_b, A_b, \Delta_b, s_0, S_b^a)$ is a LTS augmented with a set of accepting states, where $S_b^a \subseteq S_b$ is an accepting state set and the other constructs are the same as those of a LTS. A trace π is accepted by B if π passes some accepting state infinitely many times.

Parallel composition ($\|$) [21] captures the concurrent and interactive execution of LTSs. Let $L^1 = (S^1, A^1, \Delta^1, s_0^1)$ and $L^2 = (S^2, A^2, \Delta^2, s_0^2)$ be LTSs. $L^1 \parallel L^2 = (S^1 \times S^2,\ A^1 \cup A^2,\ \Delta,\ (s_0^1, s_0^2))$, where $\Delta \subseteq (S^1 \times S^2) \times (A^1 \cup A^2) \times (S^1 \times S^2)$ is computed as follows: $\Delta = \{((s^1, s^2), a, (t^1, t^2)) | (s^1, a, t^1) \in \Delta^1, (s^2, a, t^2) \in$

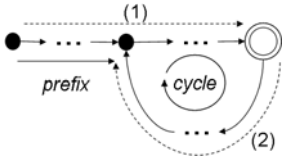

Fig. 2. Shape of Counterexample and Witness

$\Delta^2\} \cup \{((s^1, s^2), a, (t^1, s^2)) \mid (s^1, a, t^1) \in \Delta^1, a \notin A^2\} \cup \{((s^1, s^2), a, (s^1, t^2)) \mid (s^2, a, t^2) \in \Delta^2, a \notin A^1\}$.

At step 3 of the checking procedure for liveness properties, algorithms to search for strongly connected components such as nested depth-first search [14] are used by many existing model checkers (e.g. SPIN [14]). They find a counterexample that forms an infinite and lasso-shaped trace $\pi = PC^\omega$ (see Fig. 2), where the prefix P and the cycle C are finite event sequences whose subsequences contain no cycle. C passes some accepting state of $B(\neg\phi)$ depicted as a double circle in Fig. 2. Hence, we assume $\pi = PC^\omega$. An example is π^c in Table 1. A witness is a trace satisfying ϕ and is assumed to have a form $\tau = P'C'^\omega$.

4 Error Localization Procedure

This section presents an error localization technique LLL-S. The idea is that we find the closest (i.e. the most similar) witnesses to π, and then detect their differences. The inputs to LLL-S are a LTS $L = (S, A, \Delta, s_0)$, a FLTL formula ϕ where $L \not\models \phi$, and a counterexample $\pi = PC^\omega$, where $P = [a_0, a_1, \ldots, a_{m-1}]$ and $C = [b_0, b_1, \ldots, b_{n-1}]$ for $0 \leq m$ and $1 \leq n$. Let $B(\phi) = (S_\phi, A_\phi, \Delta_\phi, u_0, S^a_\phi)$. We assume that $A_\phi \subseteq A$ and a witness to be searched has a form $\tau = P'C'^\omega$.

If we consider an event as a character, a trace is regarded as an infinite string. We define the distance D between π and τ using the edit distance between finite strings on the edit operations *insertion*, *deletion* and *replacement* [20]. The edit distance between finite strings s_1 and s_2, denoted by $d(s_1, s_2)$, is the minimum cost to change one string to the other. We assume that the cost of each edit operation is 1. The distance D is defined as follows: $D(\pi, \tau) = d(P, P') + d(C, C')$.

D meets all properties of a metric, i.e. positive definiteness, symmetry and triangle inequality when we define $\pi = \tau$ iff $P = P'$ and $C = C'$. D is appropriate for our goal because it distinguishes the distance of prefixes and cycles. For example, in Table 1, $D(\pi^c, \tau^1) = D(\pi^c, \tau^2) = 1$, $D(\pi^c, \tau^3) = 2$ and $D(\pi^c, \tau^4) = 5$. D judges that both τ^1 and τ^2 are the closest to π^c while τ^3 and τ^4 are not.

4.1 Outline

As a set of witnesses is given by traces accepted by $B(\phi)$, we find every witness τ in $B(\phi)$ such that $D(\pi, \tau)$ is the smallest. In order to make τ meet the Büchi's acceptance condition (see Fig. 2), we divide the procedure to find τ into two steps: 1) finding a sequence that ends in an accepting state $s^a_\phi \in S^a_\phi$ (i.e. the

sequence (1) in Fig. 2) and 2) finding a sequence that leaves s_ϕ^a and returns to a state on the path from u_0 to s_ϕ^a (i.e. the sequence (2) in Fig. 2).

First we construct a model W_π^A from the counterexample π, embedding edit operations and their costs. W_π^A is a *Weighted Transition System* (WTS) [19], a LTS augmented with a cost function ζ_w : Transitions $\rightarrow$ Cost. As π has a structure PC^ω, W_π^A consists of a linear path corresponding to P, followed by a cycle corresponding to C. For the $P = [a_0, \ldots, a_{m-1}]$ part, states p_i and transitions (p_i, a_i, p_{i+1}) $(i = 0, \ldots, m-1)$ are generated. For the $C = [b_0, \ldots, b_{n-1}]$ part, states c_i and transitions (c_i, b_i, c_{i+1}) $(i = 0, \ldots, n-1)$ are generated, where c_n is identical to c_0. All the transitions thus generated have cost 0. The transitions are augmented by the following three types of new transitions with cost 1.

1. *Replace*: for a pair (p_i, p_{i+1}), transitions (p_i, a, p_{i+1}) where $a \in (A - \{a_i\})$, meaning replacing the event a_i with the event a. Likewise, for a pair (c_i, c_{i+1}), transitions (c_i, b, c_{i+1}) where $b \in (A - \{b_i\})$.
2. *Delete*: for a pair (p_i, p_{i+1}), transition (p_i, ϵ, p_{i+1}) meaning deleting a_i. ϵ is a null event. Likewise, for a pair (c_i, c_{i+1}), transition (c_i, ϵ, c_{i+1}).
3. *Insert*: for a state p_i, transitions (p_i, a, p_i) where $a \in A$, meaning inserting a at p_i. Likewise, for a state c_i, transitions (c_i, b, c_i) where $b \in A$.

Next, we build a product model $W_\bowtie = B(\phi) \bowtie W_\pi^A$. The problem of finding witnesses of the property that are the most similar to the counterexample is reduced to the problem of finding the shortest paths in the graph of $W_\bowtie$, starting from the initial vertex, visiting a vertex corresponding to an accepting state of $B(\phi)$ and ending in a vertex that closes the path to make a cycle. The vertex of the accepting state should be included in the cycle. We can employ a shortest path algorithm such as Dijkstra's method [7] to solve this problem. In the first step, the shortest paths from the initial vertex to the accepting vertices are obtained. Then, for each accepting vertex that has been reached from the initial vertex, the second shortest path problem is solved starting from the accepting vertex, ending in the vertices on the shortest path from the initial vertex to the accepting vertex, so as to close a cycle. Thus, we need to solve the single-source shortest path problem $v_a + 1$ times, where v_a is the number of accepting states.

The differences between τ and π indicate potential errors. LLL-S detects every difference and extracts every transition that has the erroneous event.

4.2 Constructing WTS Models

We define a WTS as an extension of a LTS [19]. A WTS is a tuple $W = (S_w, A_w, \Delta_w, q_0, \zeta, M_w)$, where S_w is a finite set of states, A_w is a set of event labels, $\Delta_w \subseteq S_w \times A_w \times S_w$ is a transition relation, and $q_0 \in S_w$ is the initial state, the total function $\zeta : \Delta_w \rightarrow \mathcal{R}$ is a *weight* to every transition, and $M_w \subseteq S_w$ is a set of *end states*. We use the terms on a LTS also for a WTS, e.g. traces.

A WTS W_π^A made from π consists of two parts: the part constructed from P and that from C which respectively show edit operations and their costs applied to P and C. Finite traces of W_π^A that pass the P and C part respectively provide P' and C'. A set of end states includes all states of the C part to indicate that a

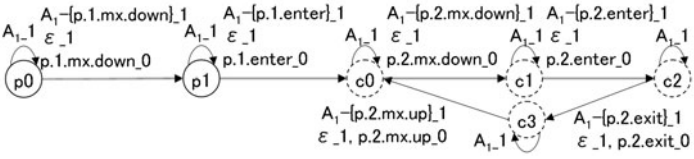

Fig. 3. WTS $W^{A_1}_{\pi^c}$ Constructed from π^c

finite trace in W^A_π ends in any element of the set, and that an accepting state of $B(\phi)$ appearing in the element is the destination of the sequence (1) in Fig. 2.

The WTS $W^A_\pi = (S_w, A \cup \{\epsilon\}, \Delta_w, q_0, \zeta_w, M_w)$ of π is constructed as follows. The state set $S_w = \{p_i | 0 \leq i < m\} \cup \{c_i | 0 \leq i < n\}$. The initial state $q_0 = p_0$ if $m \neq 0$; otherwise, $q_0 = c_0$. The transition relation $\Delta_w = \Delta_p \cup \Delta_b \cup \Delta_c$, where Δ_p, Δ_b and Δ_c are defined as follows. $\Delta_p = \{(p_i, a, p_{i+1}) |\ 0 \leq i < m-1, a \in A \cup \{\epsilon\}\} \cup \{(p_i, a, p_i) | 0 \leq i < m, a \in A\}$. $\Delta_b = \{(p_{m-1}, a, c_0) | a \in A \cup \{\epsilon\}\}$ if $m \neq 0$; otherwise, $\Delta_b = \emptyset$. $\Delta_c = \{(c_i, a, c_{i+1}) | 0 \leq i < n-1, a \in A \cup \{\epsilon\}\} \cup \{(c_{n-1}, a, c_0) | a \in A \cup \{\epsilon\}\} \cup \{(c_i, a, c_i) | 0 \leq i < n, a \in A\}$. For each $\delta \in \Delta_w$, $\zeta_w(\delta) = 0$ if either of the following conditions holds: $\delta = (p_i, a_i, p_{i+1})(i = 0, \ldots, m-2)$, $\delta = (p_{m-1}, a_{m-1}, c_0)$, $\delta = (c_i, b_i, c_{i+1})$ $(i = 0, \ldots, n-2)$, or $\delta = (c_{n-1}, b_{n-1}, c_0)$; otherwise, $\zeta_w(\delta) = 1$. The set of end states $M_w = \{c_i | 0 \leq i < n\}$.

Fig. 3 shows the WTS model $W^{A_1}_{\pi^c}$ constructed from π^c in Table 1.The initial state is p_0 and end states are states with dashed circles c_i $(i = 0, \ldots, 3)$. A weight to each transition is written after the event. The set of transitions (p, A_1, q) with weight w indicates that every transition in (p, A_1, q) has the same weight w. The P part of $W^{A_1}_{\pi^c}$ consists of the states $p_i (i = 0, 1)$ and c_0, and the transitions defined by Δ_p and Δ_b. For example, a transition $(p_0, p.1.mx.down, p_1)$ with weight 0 shows *p.1.mx.down* in P. Transitions $(p_0, A_1 - \{p.1.mx.down\}, p_1)$ mean that *p.1.mx.down* is *replaced* by another event. A transition (p_0, ϵ, p_1) represents that *p.1.mx.down* is *deleted*. Self transitions (p_0, A_1, p_0) show *insertion* operations just before *p.1.mx.down*. Likewise, the C part of $W^{A_1}_{\pi^c}$ consists of the states c_i $(i = 0, \ldots, 3)$ and the transitions defined by Δ_c.

Finite traces that pass from p_0 to c_0 present P'. Similarly, finite traces that pass from c_0 to itself via $c_i (i = 1, \ldots, 3)$ present C'.

4.3 Finding Witnesses

We next find a witness τ such that $D(\pi, \tau)$ is the smallest by conducting the single-source shortest path search twice.

We first find a sequence that ends in an accepting state $s^a_\phi \in S^a_\phi$. We compute the product of the WTS W^A_π and $B(\phi)$ so that such event sequences can be obtained by the shortest path from the initial state of the product graph.

We extend the parallel composition operation of LTSs to the operation $(\bowtie)$ of a LTS and a WTS [19]. Let $B = (S_b, A_b, \Delta_b, s_0, S^a_b)$ and $W = (S_w, A_w, \Delta_w, q_0, \zeta, M_w)$ be a Büchi automaton and a WTS such that $A_b \subseteq A_w$, respectively. Their product is a WTS $B \bowtie W = (S_b \times S_w, A_w, \Delta'_w, (s_0, q_0), \zeta', S^a_b \times M_w)$, where $\Delta'_w \subseteq (S_b \times S_w) \times A_w \times (S_b \times S_w)$ is a transition relation such that

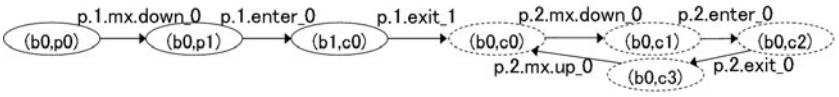

Fig. 4. Fragment of $B(\mathsf{EXIT1}) \bowtie W^{A_1}_{\pi^c}$

$\Delta'_w = \Delta^1_w \cup \Delta^2_w$ where $\Delta^1_w = \{((s_b, s_w), a, (s'_b, s'_w))|(s_b, a, s'_b) \in \Delta_b, (s_w, a, s'_w) \in \Delta_w\}$ and $\Delta^2_w = \{((s_b, s_w), a, (s_b, s'_w))|(s_w, a, s'_w) \in \Delta_w,\ a \notin A_b\}$. For each $\delta = ((s_b, s_w), a, (s'_b, s'_w)) \in \Delta'_w$, we define $\zeta' : \Delta'_w \to \mathcal{R}$ by $\zeta'(\delta) = \zeta((s_w, a, s'_w))$.

Intuitively, the WTS $B(\phi) \bowtie W^A_\pi$ labels transitions of $B(\phi)$ with costs of edit operations applied to π. Each end state $(s^a_\phi, c_M) \in S^a_\phi \times M_w$ is both an accepting state s^a_ϕ of $B(\phi)$ and an end state c_M of W^A_π. The shortest paths from the initial state (u_0, q_0) to (s^a_ϕ, c_M) present the event sequences that end in s^a_ϕ.

For each end state (s^a_ϕ, c_M), we conduct the second shortest path search to find sequences ending in a state on each shortest path from (u_0, q_0) to (s^a_ϕ, c_M). The witness $\tau = P'C'^\omega$ is finally generated by combining the sequences computed by the two shortest path searches. We remove τ whose C' is an empty sequence from candidate witnesses. We collect every witness such that the sum of the distances obtained by the first and second search is the smallest of all possible witnesses.

Consider $B(\mathsf{EXIT1})$ and $W^{A_1}_{\pi^c}$. A fragment of their product is shown in Fig. 4, where the only relevant information to find τ^1 are written. One of the shortest paths from the initial state (b_0, p_0) to an end state (b_0, c_0) presents the subsequence of τ^1 to the accepting state b_0 of $B(\mathsf{EXIT1})$: $H^1 = [$ *p.1.mx.down*, *p.1.enter*, *p.1.exit*$]$. Next, the sequence $T^1 = [$ *p.2.mx.down*, *p.2.enter*, *p.2.exit*, *p.2.mx.up*$]$ is presented by the shortest path from (b_0, c_0) to itself, which is one of the states on the shortest path from (b_0, p_0) to (b_0, c_0). We find τ^1 by combining H^1 and T^1. Another witness τ^2 is obtained using the same procedure.

4.4 Identifying Errors

To find errors in L, we compute the differences between $\pi = [a_0, a_1, \dots]$ and τ. We can assume that the different events between π and τ directly or indirectly designate causes of the property violation. If L consists of r processes, each of which is denoted by $L^h = (S^h, A^h, \Delta^h, s^h_0)$ where $0 \le h < r$ and $A = \cup_{0 \le h < r} A^h$, we identify a set of transitions over the processes triggered by the events as error candidates. However, some of the processes might not have transitions corresponding to the events to be modified. For such processes, we take a set of last transitions that occur before the differences due to the assumption that the events of these transitions trigger the events to be modified.

If an event a_d is replaced or deleted, we say that a_d is a *mismatched event*. A candidate error in L^h is its transition with the mismatched event a_d if $a_d \in A^h$; otherwise, the last transition that occurs before a_d. LLL-S returns the transition $(s, a_j, t) \in \Delta^h$ such that $0 \le j \le d$ and $a_j \in A^h \wedge \forall l \in \mathcal{N}.(j < l \le d \Rightarrow a_l \notin A^h)$.

Consider π^c and τ^2. *p.2.enter* is the mismatched event as it is replaced by *p.1.exit*. LLL-S finds error candidates for p.1, p.2 and Sema using the mismatched

event. For Sema, we have to examine the preceding events in Sema in exploring the cause of error because *p.2.enter* does not belong to the event set of Sema. The Sema's last event occurring before *p.2.enter* is *p.2.mx.down*. LLL-S reports Sema's error candidate $(1, p.2.mx.down, 2)$, which is interpreted that *p.2.mx.down* triggers *p.2.enter* of p.2. In addition, LLL-S respectively returns $(1, p.1.enter, 2)$ of p.1 and $(1, p.2.enter, 2)$ of p.2 as the other error candidates.

If an event is inserted between a_{d-1} and a_d, LLL-S reports a pair of transitions of L^h that *enclose* the inserted event as follows. 1) Return the transition of L^h with a_{d-1} if $a_{d-1} \in A^h$; otherwise, its last transition occurring before a_{d-1} using the procedure above by regarding a_{d-1} as a mismatched event. 2) Return the transition of L^h with the event a_d if $a_d \in A^h$; otherwise, its first transition occurring after a_d. LLL-S returns the transition $(s, a_j, t) \in \Delta^h$ such that $j \geq d$ and $a_j \in A^h \wedge \forall l \in \mathcal{N}.(d \leq l < j \Rightarrow a_l \notin A^h)$.

Consider finding the error candidate of Sema using τ^1. The events that enclose the inserted event *p.1.exit* in π^c are *p.1.mx.down* and *p.2.mx.down*. LLL-S returns the transitions $(0, p.1.mx.down, 1)$ and $(1, p.2.mx.down, 2)$ as a candidate cause of the violation. The inserted event *p.1.exit* may be demanded by the preceding event *p.1.mx.down* or the succeeding event *p.2.mx.down* or both.

Of all transitions computed by LLL-S, developers decide which transitions appropriately capture the erroneous behavior of L with the help of the witnesses. For example, the erroneous mutual exclusion realized by Sema is captured by the transitions given above, and both τ^1 and τ^2 show how this behavior is avoided. The presentation of witnesses and error candidates enable developers to easily identify the incorrect processes, which is an important character of LLL-S.

5 Implementation and Case Studies

We implemented a prototype tool in Java that automatically executes LLL-S. The inputs to the tool are a LTS model, the Büchi automaton of a property and a counterexample. The tool outputs a list of potential erroneous transitions and the corresponding witnesses. To enhance its performance, we have implemented some heuristics, e.g., the tool does not conduct the second search in Section 4.3 if the edit distance obtained by the first search is larger than the smallest value of distance D computed in previous iterations. The tool also supports error localization for safety property violation, which produces finite counterexamples [9]. Since the cycle of a counterexample, in this case, is regarded as an empty sequence $[\epsilon]$, we revise the way of synthesizing the WTS model in Section 4 so that ϵ can be replaced by another event [19]. The witnesses to be searched are infinite and lasso-shaped because they satisfy the Büchi's acceptance condition.

We conducted seven case studies with the prototype tool: the microwave oven (MOvn) [5], the Andrew File System (AFS-1) [26], CSys, the mine pump (MPmp) [25], and the distributed databases (DDb1, DDb2 and DDb3) [21]. Each case study was conducted as follows: 1) we made a LTS model consisting of one or more processes, 2) we prepared a FLTL property that the model did not satisfy, 3) we obtained a counterexample and a Büchi automaton recognizing the property using the model checker LTSA [21], and 4) we executed our tool and

Table 2. The Number of Generated Witnesses Indicating Errors (A) out of Total Number of Generated Witnesses (B) and Execution Time for Each Case

System		Büchi Automaton		Counterexample	Witnesses		Time
Model	States/Trans.	Property	States/Trans.	Prefix/Cycle	(A)	(B)	[s]
MOvn	7/21	HEAT	7/91	0/4	2	10	0.23
AFS-1	16/21	VALID	4/28	5/-	2	8	0.05
CSys	16/32	MUTEX	4/16	4/-	7	7	0.04
		EXIT2	6/99	5/4	3	12	0.34
MPmp	22/56	EMG	2/30	3/4	2	9	0.16
DDb1	160/402	QUIS	10/897	12/1	2	23	0.25
DDb2	6460/18537	QUIS	10/890	26/33	10	40	5.35
DDb3	-	SAFE	452/33900	18/-	57	467	37.14

manually investigated whether its result contained the transitions that were the causes of violations and the witnesses that appropriately explained the causes or not. Table 2 shows the results of the case studies. When the target property is a safety property, the length of the counterexample cycle is written as "-" in the table. We executed our tool ten times for each case on 3.4GHz Pentium 4 with 2GB RAM (JDK 1.6.0), and its average is shown as execution time. The DDb3 model has more than 2 million states and 60 million transitions, but its size could not be computed due to the heap memory limitation (shown as "-").

MOvn and MPmp are models with a single process. Although in the case of MOvn, we had to change the shape of the counterexample beforehand because LLL-S generates witnesses based on its shape, LLL-S successfully pointed out transitions that include the erroneous ones in both cases. Compared to manual error search, LLL-S made the search space for errors reduced. AFS-1, CSys, DDb1, DDb2 and DDb3 models consist of multiple processes. To find errors in component processes by hand, we have to investigate the behavior of all processes. The composite behavior analysis of all processes requires a complex composite model and makes it hard to manually identify errors whose cause is rooted in concurrency. LLL-S generated error candidates for all the processes we investigated and in all the cases, real errors were located from its subset. The task of examining the error candidates saves us much effort in locating errors compared to the case of analyzing the counterexample without any other clues.

Let us see the DDb1 case. It consists of a ring of three database nodes and a controller that allows a single update of the local data of each node. QUIS requires that every node become inactive, i.e., each node is not engaged in an update [21]. LLL-S found error candidates and the corresponding witnesses showing that an inactive node should not update or the controller should not terminate. LLL-S also reported the appropriate cause of violation that the controller terminates before checking inactivity of all nodes. We selected the error of the controller guided by two witnesses indicating how its incorrect behavior was avoided.

We next investigated how execution time of our tool respectively scaled according to the size of the Büchi automaton, and the prefix and cycle length of the counterexample using the MPmp case (shown in Table 3 and Fig. 5).

Table 3. Execution Time vs. Büchi automaton Size

Büchi Automaton	States	4	8	11	14	29	29	35	35	52	50	64	64
	Trans.	58	176	216	302	534	650	751	855	931	1190	1327	1471
Time [s]		0.16	0.69	0.40	0.66	0.35	1.13	0.97	1.42	0.60	2.59	1.00	1.05

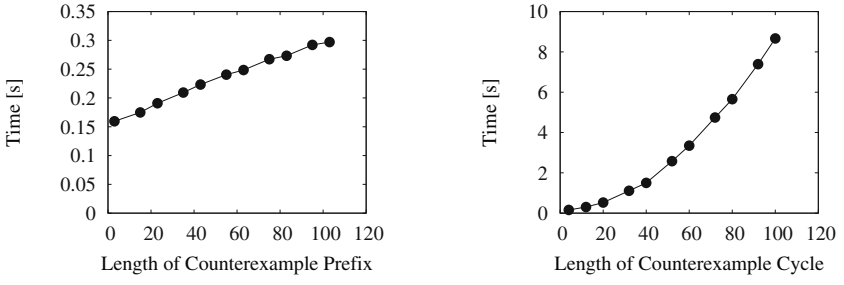

Fig. 5. Execution Time vs. Length of Counterexample Prefix (*left*) and Cycle (*right*)

In Table 3, each Büchi automaton was made by adding safety properties to EMG that the model satisfied. In Fig. 5, we expanded the cycle of the counterexample in Table 2 to make longer counterexamples to be used as samples of different size. Both results indicates that LLL-S practically handles large Büchi automata, and counterexamples with long prefixes or cycles. The execution time for large automata is almost the same as that for medium-sized ones due to the heuristics explained at the beginning of this section. Fig. 5 shows that the cycle length of a counterexample has a larger impact on the execution time of LLL-S than its prefix length. This is because the cycle length influences on the running time of the first search in Section 4.3 as well as the second search, whereas the prefix only influences on the first search.

6 Discussions and Related Work

Computational Complexity. We estimate the time to find witnesses using LLL-S. Let the counterexample $\pi = PC^\omega$ where $|P| = m$ and $|C| = n$, and the Büchi automaton of the property ϕ be $B(\phi) = (S_\phi, A_\phi, \Delta_\phi, u_0, S^a_\phi)$ where $|S_\phi| = v_\phi$ and $|S^a_\phi| = v^a_\phi$. WTS W^A_π has $m+n$ states and n end states, and $B(\phi) \bowtie W^A_\pi$ has $v_\phi(m+n)$ states and $v^a_\phi n$ end states. The first shortest path search in Section 4.3 requires $O(v_\phi(m+n)\log(v_\phi(m+n)))$ time. The second search is conducted $v^a_\phi n$ times because a source of the search is an end state of $B(\phi) \bowtie W^A_\pi$. Each search is conducted on the subgraph of the product consisting of $v_\phi n$ states because only the C part of W^A_π is used for the search. For each end state, a shortest path search requires $O(v_\phi n \log(v_\phi n))$ time. Thus, the total time of the second search is $O(v^a_\phi v_\phi n^2 \log(v_\phi n))$. If $m \approx n$, the running time is dominated by the total time of the second search. Thus, LLL-S requires $O(v^a_\phi v_\phi n^2 \log(v_\phi n))$ time.

On Fairness Constraints. When we verify a liveness property on a LTS, we often assume a kind of fairness constraints, *fair choice* [10]. Fair choice asserts that if a choice over a set of transitions is executed infinitely often, every transition in the set will be executed infinitely often. Model checking with fair choice finds an infinite and lasso-shaped counterexample under the constraint, which is the same assumption of LLL-S. Thus, LLL-S is applicable to the case.

On Property Patterns. It is useful to investigate what kinds of witnesses LLL-S produces for each property pattern [8]. Some of the liveness properties used in our case studies are written in the response pattern formula $\mathbf{G}(p \Rightarrow \mathbf{F}q)$ [8]. In this case LLL-S generated two kinds of witnesses: witnesses in which p never holds, or in which q holds after or at the same time as p holds. For example, the witnesses for π^c are classified into either those in which p.1 never enters the critical region, or those in which p.1 leaves the critical region after entering there. Although developers need to identify the appropriate ones out of all found witnesses, this information may enable them to narrow down the candidate errors.

Related Work. We previously proposed a method to find behavior model errors with infinite counterexamples [19]. Although it finds the witnesses that resemble the counterexample analogous to LLL-S, it is not based on a solid criterion to measure distances between infinite traces and may miss witnesses that appropriately point out errors. LLL-S solves the problem using the distance D.

J. Beer *et al.* [2] proposes a way to explain counterexamples for LTL model checking. While its goal is not error localization, it complements LLL-S.

Our work is related to the debugging techniques for programs as a result of model checking. A way to identify C program errors and their causes was developed by Groce and Visser with multiple counterexamples leading to the same error state [13], and later by Groce *et al.* with a single counterexample [12]. Chaki *et al.* extends the work [12] to abstracted programs [3]. Griesmayer *et al.* proposed an error localization technique for C programs [11]. Ball *et al.* proposed a technique to isolate causes of errors using counterexamples [1].

In software testing, Zeller proposed a way to find the cause-effect chains of errors in C programs [27]. Cleve and Zeller later developed a complementary technique that identifies when failure causes propagate to faults [6]. Spectrum-based fault localization techniques collect faulty runs and correct runs and compare them with certain criteria to locate faults in programs [15,23].

The above approaches resemble ours in that the comparison of a faulty run with a correct run tells us errors where the correct run is the closest to the faulty run based on the specific distance between finite runs. However, even if these distances are adapted to our context, they do not distinguish between prefixes and cycles of infinite runs and cannot overcome the problem discussed in Section 2. Although Chaki *et al.* [3] tackles the error localization problem of liveness properties, their technique reduced the problem to the SAT, which is NP-complete. LLL-S solves the classical graph search problem and performs much more efficiently. Finally, the existing methods [13,12,1,3,27,6] assume the existence of at least one correct run. LLL-S uses Büchi automata to build witnesses and does not require any correct runs supplied by the user.

Killian *et al.* developed a model checker for C++ programs, MaceMC, and its debugger MDB [16]. MaceMC supports verification of liveness properties. MDB helps developers understand errors by returning a comparison of a faulty run obtained by MaceMC with a correct run which shares a common prefix. The idea resembles ours, but only focuses on a certain kind of liveness properties.

Mohri [22] and Konstantinidis and Silva [17] developed graph-based methods that compute the edit distance between finite regular languages. LLL-S focuses on infinite strings, whose similarity cannot be computed by their methods.

7 Conclusions

In this paper, we have presented a novel automated technique to locate errors in behavior models based on the result of fluent model checking. We adopt a counterexample-based and model-based approach, which require only the model composition and classical graph search techniques. In particular, we can generate infinite-length witnesses that fix the given infinite counterexample to satisfy the property, which, we believe, is a major breakthrough.

There is much future work including integration of fluent model checking [9] with LLL-S, further practical case studies and generation of domain-specific witnesses. The last issue extends our work to help developers fix model errors [18]. Since witnesses are searched on Büchi automata, they do not reflect knowledge of the whole range of the problem domain. One of the possible solutions to this problem is to introduce the properties that hold in the target model. The introduced properties are formal descriptions of the domain knowledge.

References

1. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: POPL 2003, pp. 97–105 (2003)
2. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining counterexamples using causality. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 94–108. Springer, Heidelberg (2009)
3. Chaki, S., Groce, A., Strichman, O.: Explaining abstract counterexamples. In: SIGSOFT 2004/FSE 12, pp. 73–82 (2004)
4. Clarke, E.M.: The birth of model checking. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 1–26. Springer, Heidelberg (2008)
5. Clarke, E.M.J., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999)
6. Cleve, H., Zeller, A.: Locating causes of program failures. In: ICSE 2005, pp. 342–351 (2005)
7. Dijkstra, E.W.: A note on two problems in connection with graphs. Numerische Mathematik, 269–271 (1959)
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE 1999, pp. 411–420 (1999)
9. Giannakopoulou, D., Magee, J.: Fluent model checking for event-based systems. In: ESEC/FSE 2003, pp. 257–266 (2003)

10. Giannakopoulou, D., Magee, J., Kramer, J.: Checking progress with action priority: Is it fair? In: ESEC/FSE 1999, pp. 511–527 (1999)
11. Griesmayer, A., Staber, S., Bloem, R.: Automated fault localization for C programs. Elec. Notes in Theor. Comp. Sci. 174, 95–111 (2007)
12. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. STTT 8(3), 229–247 (2006)
13. Groce, A., Visser, W.: What went wrong: explaining counterexamples. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 121–135. Springer, Heidelberg (2003)
14. Holzmann, G.J.: The SPIN model checker: primer and reference manual. Addison-Wesley, Reading (2004)
15. Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: ICSE 2002, pp. 467–477 (2002)
16. Killian, C., Anderson, J.W., Jhala, R., Vahdat, A.: Life, death, and the critical transition: finding liveness bugs in systems code. In: NSDI 2007, pp. 243–256 (2007)
17. Konstantinidis, S., Silva, P.V.: Computing maximal error-detecting capabilities and distances of regular languages. Technical report, CMUP 2008-28 (2008)
18. Kumazawa, T., Tamai, T.: Iterative model fixing with counterexamples. In: APSEC 2008, pp. 369–376 (2008)
19. Kumazawa, T., Tamai, T.: Localizing errors and presenting alternatives: a model-based approach. In: SES 2009, pp. 55–62 (2009) (in Japanese)
20. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady 10(8), 707–710 (1966)
21. Magee, J., Kramer, J.: Concurrency: state models & Java programming, 2nd edn. John Wiley & Sons, Chichester (2006)
22. Mohri, M.: Edit-distance of weighted automata: general definitions and algorithms. Int. J. of Found. of Comp. Sci. 14(6), 957–982 (2003)
23. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: ASE 2003, pp. 30–39 (2003)
24. Schmidt, D.C.: Model-driven engineering. IEEE Computer 39(2), 25–31 (2006)
25. Uchitel, S., Brunet, G., Chechik, M.: Synthesis of partial behaviour models from properties and scenarios. IEEE TSE 35(3), 384–406 (2009)
26. Wing, J.M., Vaziri-Farahani, M.: A case study in model checking software systems. Sci. of Comp. Prog. 28, 273–299 (1997)
27. Zeller, A.: Isolating cause-effect chains from computer programs. In: SIGSOFT 2002/FSE 10, pp. 1–10 (2002)