Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2013, Article ID 138287, 10 pages
http://dx.doi.org/10.1155/2013/138287

Research Article

A Latent Implementation Error Detection Method for Software Validation

Jiantao Zhou,[1] Jing Liu,[1] Jinzhao Wu,[2] and Guodong Zhong[1]

[1] *College of Computer Science, Inner Mongolia University, Hohhot 010021, China*
[2] *Guangxi Key Lab of Hybrid Computation and IC Design Analysis, Guangxi University for Nationalities, Nanning 530006, China*

Correspondence should be addressed to Jiantao Zhou; zhoujiantao@tsinghua.org.cn

Received 4 February 2013; Accepted 18 February 2013

Academic Editor: Xiaoyu Song

Model checking and conformance testing play an important role in software system design and implementation. From the view of integrating model checking and conformance testing into a tightly coupled validation approach, this paper presents a novel approach to detect latent errors in software implementation. The latent errors can be classified into two kinds, one is called as Unnecessary Implementation Trace, and the other is called as Neglected Implementation Trace. The method complements the incompleteness of security properties for software model checking. More accurate models are characterized to leverage the effectiveness of the model-based software verification and testing combined method.

1. Introduction

In software engineering practices, model-based software development and analysis methods receive extensive attention [1]. Software model checking [2] and model-based conformance testing [3] are two well-established approaches validating the accuracy of software executions. Model checking aims at verifying whether the software specification model satisfies a set of key properties that represent the software functional requirements, while conformance testing aims at checking if the actual black-box implementation behaves as what the specification model describes according to some kind of conformance relation. More specifically, software model checking validates the specification model with the key properties during the design phase, while conformance testing checks the nonconformance relation between the system programs and the specification model during the implementation phase. Thus, model checking and conformance testing could perform sequentially and work as an integrated validation process to assure the functional correctness of a software system.

However, only applying model checking followed by conformance testing is not fully satisfactory. The essential reason is that model checking focuses on an accurate system formal model, not considering the system implementation, while conformance testing focuses on checking whether the system implementation behaves as the model specified, not considering whether the key properties are totally tested [4]. Specifically, in the ideal scenario, all the software behaviors in the system implementation should satisfy the key properties, that is, all key properties are tested and all system behaviors are verified, as shown in the midpart of Figure 1. Unfortunately, in the software engineering practices, there always exist some key properties verified by model checking in the design phase, but they might not be tested at all in the implementation phase. This kind of scenario is called as "Under Tested," as shown in the left part of Figure 1. From the other side, some system error behaviors may still exist in the software implementations after conformance testing, and they also might not be checked out by model checking in the design phase. This kind of scenario is called as "Under Verified," as shown in the right part of Figure 1. Therefore, the "Under Tested" and "Under Verified" scenarios are two major drawbacks of the traditional software validation methods, where the model checking and the conformance testing exercise sequentially and individually.

In order to integrate model checking and conformance testing into a tightly coupled validation approach, two kinds

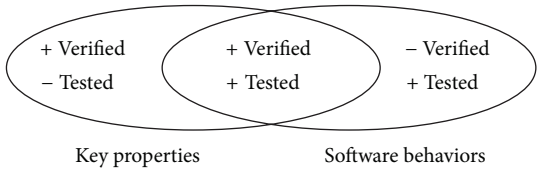

Figure 1: Three kinds of effects for the traditional software validation method.

of studies have been done. First, most studies in the literature focus on fixing the "Under Tested" problem [4, 5]. Specifically, the key properties are considered in the conformance test case generation method, so executing such set of test cases could guarantee that all key properties are tested. For example, key properties could be formulized as the test purpose models and the test generation process utilizes model checking as a major generation technology. Consequently, all generated test cases will definitely cover the key properties. These studies could improve the "Under Tested" scenario, and the major contribution is to make the conformance testing more complete and more accurate, that is, the software verification well complements the software testing. However, we also need certain studies to improve the "Under Verified" scenario, that is, the software testing should complement the software verification as well. But in the literature, this kind of studies is really few. Black-box checking [6] and adaptive model checking [7] are two special cases. They aim to construct more specific system models from a partial or an empty model through testing and learning. The difficulties in this kind of studies result from the fact that the implementation errors are always detected unexpectedly through the conformance testing process, that is, we cannot guarantee detecting such errors definitely. Therefore, we herein propose a more proactive method to detect such latent errors which have been programmed in the system implementations and improve the "Under Verified" scenario consequently.

To demonstrate the motivation of our work more clearly, we make further analysis first. Without loss of generality, we adopt an instance model checking method based on the Input-Output Labeled Transition System (IOLTS) models [4], and an instance conformance testing approach based on the IOLTS models and the Input-Output Conformance (IOCO) relation [8]. Implementing an automatic vending machine (AVM) system is taken as an example. It is supposed that a specific software implementation of the vending machine has been developed. The machine releases the tea or milk when a coin is inserted. But in this implementation, there exists a fatal error, that is, this machine could also release the tea or milk as an incorrect coin is inserted. Furthermore, it is supposed that the formal model of this AVM system has no corresponding parts to deal with such exception behavior. Therefore, as discussed previously, if we do not consider a specific property towards this exception error in the model checking phase unintentionally, the verification will pass without counterexamples. Then, some test cases are generated from this verified model, and no test case aims to detect such exception behavior, because no corresponding specification exists in the verified model. So, according to the IOCO relation, the conformance testing will also determine the conformance relation between the implementation and the model. Herein, a serious problem is emerged. Though this exception behavior does exist in the system implementation, both model checking and conformance testing do not detect this fatal error. However, the error should be repaired. Based on this analysis, a novel Latent Implementation Error Detection (LIED) method for the software validation will be proposed in this paper to proactively detect such latent errors in the implementations. The LIED method explores model checking towards the actual software implementation to check such latent errors and utilize the counterexamples to improve the system models. The LIED method not only complements the incompleteness of the key security properties for the software model checking, but also constructs more accurate models to promote the effectiveness of the model-based software verification and testing sequentially combined method. On the one hand, the original model checking could be performed more completely, because the key properties and the system models are both well improved. On the other hand, the conformance testing could be performed more precisely, because the improvement of the system models results in better test cases, which are generated with higher accuracy and stronger capability of detecting implementation errors. So, the "Under Verified" scenario is well improved consequently.

The paper is organized as follows. Firstly, certain preliminaries and related work are discussed in Section 2. Then, two kinds of specific latent implementation errors are defined formally in Section 3. Finally, the LIED method is given in detail in Section 4. The method includes three major parts: enumerating the possible key properties, model checking towards the system implementation, and revising the system model using the counterexamples. To elaborate the feasibility and effectiveness of the LIED method, we put it into practice with a simplified AVM system as a representative.

2. Preliminaries and Related Work

In this section, we first introduce the formal definition of the IOLTS model and the basic ideas about the IOCO relation-based conformance testing method. Then, we discuss several related and important studies on how to integrate model checking and conformance testing technologies in recent literature.

An IOLTS model is actually an LTS model with explicitly specified input and output actions [8]. It is widely used not only as a kind of formal modeling approach to model the reactive software systems directly, but also as the operational semantic model for several process languages, such as LOTOS (Language of Temporal Ordering Specification).

Definition 1. An IOLTS model is a four-tuple IOLTS = (Q, L, T, q_0), where

(1) Q is a countable, nonempty set of states;

(2) $L = \mathrm{LI} \cup \mathrm{LU} \cup \mathrm{LE}$: LI is a countable sets of input action labels, LU is a countable sets of output action

labels, LE is a countable sets of inner action labels, and they are disjoint;

(3) $T \subseteq Q \times L \times Q$, is the action transition relation;

(4) q_0 is the initial state.

Definition 2. For an IOLTS model M,

(1) $q \overset{\sigma}{\Rightarrow} =_{\text{def}} \exists q' : q \overset{\sigma}{\Rightarrow} q'$, where $\sigma \in L^*$, that is, $\exists q \xrightarrow{t_0} q_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} q' : t_i \in L(0 \le i \le n)$;

(2) Traces $(M) =_{\text{def}} \{\sigma \in L^* \mid q_0 \overset{\sigma}{\Rightarrow}\}$.

An IOLTS model for the specification of the AVM software is presented in Figure 2(a). In this specification, when a coin is inserted, which is modeled as an input action "?c," the machine will release a bottle of tea or milk, which is modeled as two output actions "!t" and "!m." We also present two IOLTS models for the AVM system implementations. The model $i1$ in Figure 2(b) specifies an implementation with part functions of the AVM system, that is, when a coin is inserted, only a bottle of tea is released. The model $i2$ in Figure 2(c) specifies an implementation with additional functions of the AVM system, that is, this machine may release a bottle of milk after an incorrect input action "?k," that is, no coin is inserted actually.

The IOCO relation-based testing approach has well-defined theoretical foundation [9] and high feasibility with automatic testing tools, such as Torx [10] and TGV [11]. The major framework of this testing approach has four related components. First, IOLTS is used to model the specification of a software system, and then IOLTS which is input enabled is further defined as IOTS (Input Output Transition System) to specify the behavior model of the system implementation. IOTS models characterize significant external observations in conformance testing, for example, quiescent states are crucial for distinguishing valid no-output actions from deadlocks. It just represents the state that an implementation is waiting for an input data. Second, the IOCO relation is defined as follows.

Definition 3. i IOCO $s \Leftrightarrow_{\text{def}}$ for all $\sigma \in$ traces(s): out$(i$ after $\sigma)$ $\subseteq$ out$(s$ after $\sigma)$, where i is an implementation of the system specification s.

Its intuitive idea is to compare output, that produced after executing trace σ in i and s, respectively, where σ is generated just from s. Two major aspects are emphasized in the definition: what actions should be observed, and what it means for an implementation to conform to its specification. Taking the AVM system in Figure 2 as an example, according to the IOCO relation definition, it is determined that $i1$ IOCO s and $i2$ IOCO s, that is, the implementations with part functions or additional functions are successfully determined to conform to the specification model. IOCO relation is a guidance of test case generation. Third, test cases are generated automatically and recursively based on specification models. Generally, the first transition of a test case is derived from the initial state of the specification model, after which the remaining part of the test case is recursively derived from all reachable states. Traces are recorded from the initial state to reachable states including quiescent states. If an output action, whether a real output or quiescence, is not allowed in the specification model, the test case will terminate with fail, otherwise, continue the further trace explosion or just terminate with pass at last. The detailed test generation algorithm is presented in [8]. Finally, the test execution is an asynchronous communication, directed by a test case, between the system implementation and its external environment, that is, a tester. The tester provides the implementation with input data and then observes its responses, a real output or just quiescence. If the fail state is reached where the real observations have not been prescribed in the test case model, the nonconformance between this implementation and its specification is definitely determined. Test cases are sound if they are able to make this kind of nonconformance decision.

As for related studies about integrating model checking and conformance testing technologies in the literature, most of them focus on fixing the "Under Tested" problem, which is mentioned in the above section. That is, key properties are involved in the test case generation methods, so executing such set of test cases could guarantee that all key properties are tested. The VERTECS research team at INRIA [4, 12–14] specifies certain key properties, such as possibility properties ("something good may happen") and safety properties ("something bad ever happens"), using the IOLTS-based formal models, and integrates these property models directly into the IOCO test case generation algorithm. Besides, other studies [5, 15–18] first formulize the key properties as the test purpose models and then perform model checking to generate the test cases. Finally, the produced counterexamples, actually representing the system execution traces satisfying the key properties, could be acted as the conformance test cases. Consequently, all generated test cases will definitely cover the key properties. The related studies mentioned previously could improve the "Under Tested" scenario and make the conformance testing more complete and more accurate. However, black-box checking [6] and adaptive model checking [7] are two special cases which aim to construct more specific system models through testing and learning, where initially the system implementation is available but no specific model or just only a partial model is provided. The model refinement process, which consists of the model checking and test execution, is performed iteratively to produce more accurate semimodels which conform to the system implementations, until the model satisfies the required key properties and conforms to the system implementation. In this paper, we propose an LIED method to proactively detect certain latent errors in the implementations and improve the "Under Verified" scenario consequently. The LIED method not only complements the incompleteness of key security properties for the software model checking, but also constructs more accurate models to promote the effectiveness of the model-based software verification and testing sequentially combined method.

Based on the discussion in Section 1 and previous related work analysis, we should note that integrating the software verification and conformance testing needs definitely iterative refinements, as shown in Figure 3. Traditional software

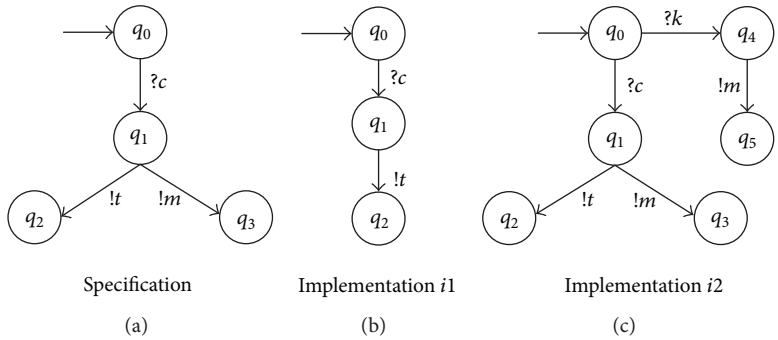

Figure 2: The specification and implementation IOLTS models for the AVM software.

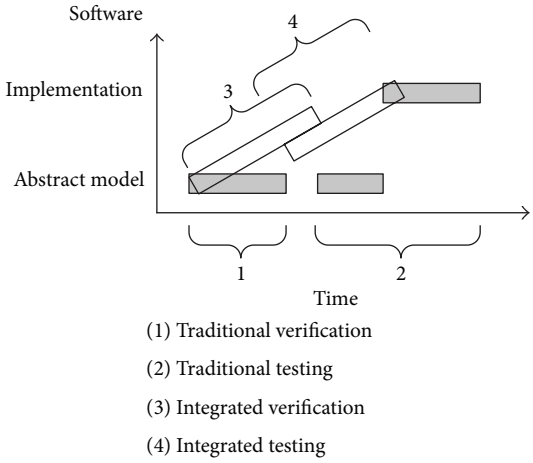

Figure 3: The iterative refinements in the integrated verification and testing method.

verification and conformance testing execute sequentially and separately, and they often have obvious boundaries towards the abstract models or the implementation. Especially, the model-based test case generation process will use abstract models. However, in the integrated software verification and testing methodology, these two methods are performed iteratively and complementarily. First, the model checking and testing process have no strict boundaries, and several semimodels and semiimplementations may exist before the final system model and implementation are developed. For example, conformance testing can be performed ahead on some semiimplementations to guide the refinement of semimodels. After iterative refinements, the final system model and system implementation are destined to be more accurate and less error-prone. Second, some specific model checking technologies could be used in the testing phase complementarily and vice versa. For example, CTL model checking algorithm is used to generate test cases [5].

In this paper, the LIED method is regarded as an integrated software validation method. It is essentially well compatible with the traditional model checking and conformance testing procedures. That is, the LIED method is developed as a complementary method for detecting latent implementation errors, not for replacing the traditional model checking and conformance testing. So, the LIED tends to be an accelerant, because its central merit is to construct more accurate system formal models, which are quite helpful to promote the effectiveness of model checking and the model-based conformance testing. We could perform model checking, the conformance testing, and the LIED process iteratively and complementarily. Consequently, these validation methods work collaboratively to make software design and implementation more effective and more efficient.

3. Two Kinds of Latent Implementation Errors

As discussing the motivation of this paper in Section 1, we demonstrate that though both model checking and conformance testing have been performed successfully, some kinds of programmed errors still exist in the software implementations. That is, these two software validation methods are ineffective against detecting such latent errors. Therefore, we propose an LIED method to proactively detect such latent errors herein.

Before presenting the detailed LIED method, we start with formally defining the specific kinds of latent implementation errors that we want to detect and repair in this paper. The concept of Trace in the IOLTS modeling (refer to Definition 2) is used to specify such implementation errors, that is, any specific error will definitely correspond to an execution trace in the IOLTS model for the system implementation. So, we could suppose with certain rationality that as long as the system implementation is programmed with some latent errors, there definitely exists an execution trace, in the IOLTS model of the system implementations, describing how such error behaviors execute step by step. Herein, we focus on two specific kinds of latent implementation errors: unnecessary implementation error and neglected implementation error.

Definition 4 (unnecessary implementation trace (UIT)). For an IOLTS MS (software specification model) and an IOLTS MI (software implementation model), where

$Q(\text{MS}) \cap Q(\text{MI}) = \Phi$, LI(MS) = LI(MI), and LU(MS) = LU(MI), and key property set P (temporal logic formulae):

UIT (MI) $=_{\text{def}}\{\sigma \in \text{obvTraces}(M_I) - \text{obvTraces}(M_S) \mid \exists p \in P : \neg(\sigma \mapsto p)\}$, where obvTraces $(M)=_{\text{def}}\{\sigma \in (L_I \cup L_U)^* \mid \sigma = \sigma'|_{L_I \cup L_U}, \sigma' \in \text{Traces}(M)\}$.

The obvTraces operator restricts the original traces σ' for an IOLTS model to only input and output action labels $(L_I \cup L_U)^*$, that is, the inner action labels are omitted and just externally observed actions are considered. So, the intuitive idea of the Unnecessary Implementation Trace consists of two parts. On the one hand, such traces could be observed from the system implementation, but not described in the system specification models, that is, $\sigma \in \text{obvTraces}(M_I) - \text{obvTraces}(M_S)$. On the other hand, such traces do not satisfy the key properties which the system functional requirements desire, that is, $\exists p \in P : \neg(\sigma \mapsto p)$. In a word, every UIT trace models a detailed unnecessary implementation error that is programmed in the system implementation, and the system specification model has no corresponding parts to deal with such exception behaviors.

Clearly, in this case, if we do not consider a specific key property against this unnecessary implementation error in the model checking phase, the verification does pass. After then, the IOCO conformance testing will also pass, because none of test cases, which are generated from verified system model, are capable of detecting such implementation error. Consequently, though an unnecessary implementation error does exist in the system implementation, both model checking and conformance testing do not detect this fatal error, and this kind of errors indeed should be repaired.

Definition 5 (neglected implementation trace (NIT)). For an IOLTS MS (software specification model) and an IOLTS MI (software implementation model), where $Q(\text{MS}) \cap Q(\text{MI}) = \Phi$, LI(MS) = LI(MI), and LU(MS) = LU(MI), and key property set P (temporal logic formulae):

NIT (MI) $=_{\text{def}}\{\sigma \in \text{obvTraces}(M_I) \cap \text{obvTraces}(M_S) \mid \exists p \in P : \neg(\sigma \mapsto p)\}$. Each Neglected Implementation Trace represents a specific neglected implementation error, but this kind of traces is a really special case. First, they appear in the specification models, as well as the implementation models, that is, $\sigma \in \text{obvTraces}(M_I) \cap \text{obvTraces}(M_S)$. Besides, they do not satisfy the key properties which the software functional requirements desire, that is, $\exists p \in P : \neg(\sigma \mapsto p)$. Thus, under normal circumstances, the model checking phase can detect such exceptional behaviors. However, under special circumstances, the Neglected Implementation Trace may be omitted by the abstraction in the model checking procedure. In this case, the verification will be passed unexpectedly. Then, we take the conformance testing into account. Because such neglected implementation traces behave the same in both MS and MI, the IOCO conformance relation between the specification model and the implementations are still determined unexpectedly. That is, we consider the exception behaviors during the IOCO conformance testing as legal behaviors, because they have been specified in the MS in the same way. Consequently, though the neglected implementation error does exist in the system implementation and specification, both model checking and conformance testing do not detect this fatal error, and this kind of errors indeed should be repaired too.

To sum up, we aim to detect and repair two kinds of latent implementation errors, that is, the unnecessary implementation errors and the neglected implementation errors, which may not be detected using the traditional model checking and IOCO conformance testing combined method. They have the same fatal effects, but they result from different causes. Therefore, in the LIED method, we utilize the unified method to detect these two kinds of latent implementation errors, but fix them using respective methods.

4. The LIED Method

As we discussed in Section 1, the unnecessary implementation error and the neglected implementation error always occur nondeterministicly through the conformance testing process, that is, we cannot guarantee detecting such errors definitely. Therefore, the LIED method is designed to proactively detect such latent errors which have been programmed in the system implementations. In this section, we first present the central idea and the framework of our LIED method. Then two core parts of this method are discussed in detail, respectively, that is, constructing the analogy set for the key properties and improving the system formal models and system implementations with the counterexamples. Finally, the AVM system is analyzed using the LIED method as a representative to elaborate the feasibility and effectiveness of the LIED method.

4.1. The Overarching Methodology. The central goal of our LIED method is to find out some unnecessary implementation errors or neglected implementation errors, where no evident clues are provided by the system models. So, designing the LIED method has two necessary preconditions. First, we need to check the system implementation. Second, we need to check as many key properties as possible against the system implementations. Consequently, we adopt model checking as the basic technology and apply it directly into the system implementations. That is, the central idea of our LIED method is to explore model checking towards the actual software implementation to check whether some kinds of latent implementation errors exist and then utilize the counterexamples, which illustrate the exception behavior executions, as guidance to improve the system models and implementations.

The framework of the LIED method is shown in Figure 4. It is composed of three related parts.

(a) *Constructing the Analogy Set for Key Properties*. The traditional model checking is performed against the original set of key properties. Such key properties are usually extracted from the functional requirement specifications for a specific software system, and they describe the necessary functional system behaviors. However, in order to detect more latent

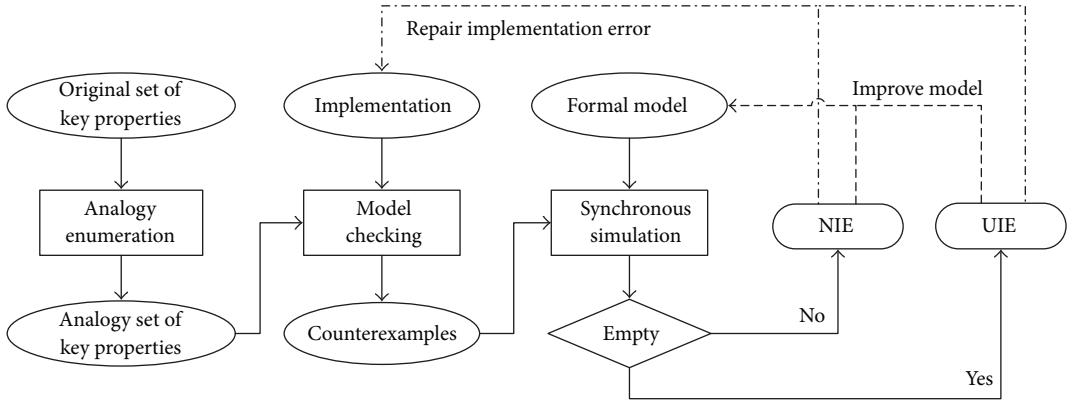

Figure 4: The framework of the LIED method.

implementation errors, we need to do some analogy enumeration based on the original set of key properties. For example, an original key property could be specified as "if condition is true then do actions." The analogy enumeration for this property could construct two more properties, that is, "if condition is false then do actions" and "if condition is true then not do actions." In this way, after such analogy enumeration procedure, the analogy set of key properties are generated for checking more possible exception behaviors for the system implementations. The details of the analogy enumeration procedure are discussed in Section 4.2.

(b) *Detecting the Latent Implementation Errors*. Based on the analogy set of key properties, we apply model checking into the system implementations directly, and the Copper model checker [19] is adopted in this paper. In this way, we could proactively detect certain latent implementation errors against the analogy set of key properties, and such latent errors are actually not detected in the traditional software verification process. If a counterexample is produced, we may detect a latent implementation error, and this counterexample could be used as an intuitive guidance for improving the system models and the system implementations.

(c) *Revising the Models or Implementations with Counterexamples*. In this paper, we propose a counterexample-guided refinement method for the software validation process. Specifically, we perform synchronous simulation between a specific counterexample and the system model. If the simulation produces an empty set of synchronous traces, it means that the exception behaviors which are represented by this counterexample are not considered in system model; an unnecessary implementation error, that is, UIE, is actually detected. In this case, the system model should be improved with additional parts about dealing with the exception behaviors, and the system implementation should be fixed too. Otherwise, if the simulation produces a nonempty set of synchronous traces, it means that though the system model has corresponding parts to deal with such exception behaviors, a neglected implementation error, that is, NIE, is still detected from the system implementation. In this case, the system model should be modified according to the counterexample scenario, and the system implementation should be fixed too. The details of the synchronous simulation procedure and the corresponding refinements are discussed in Section 4.3.

The LIED method is developed for detecting latent implementation errors, and it is well compatible with traditional model checking and conformance testing procedures. The major advantages of our LIED method lie in two aspects. First, it complements the incompleteness of the key properties for the software validation. More importantly, it benefits constructing a more accurate system formal model to promote the effectiveness of the model-based software verification and testing sequentially combined method. Specifically, the original model checking could be performed more completely, because the key properties and the system models are both improved, and the conformance testing could be performed more precisely, because the improvement of the system models results in better test cases, which are generated with higher accuracy and stronger error-detecting capability. Consequently, the LIED method improves the "Under Verified" scenario as expected.

4.2. Constructing the Analogy Set for Key Properties. In order to detect more latent implementation errors proactively and focus on more necessary functional behaviors for a software system, we perform analogy enumeration based on the original set of key properties, which are constructed in the traditional model checking phase. According to the survey of patterns in property specifications [20–22], most properties (more than 90%) could be formulized within five kinds of property patterns, where each of them could be either specified as a Linear Temporal Logic (LTL) formula or a Computation Tree Logic (CTL) formula. As shown in Table 1, we present the original specifications (OS) and its analogy set (AS) for each kind of property patterns, respectively.

As we want to detect more latent implementation errors proactively, the above analogy set for properties is used from two aspects in the LIED method. On one hand, if one specific property is verified in original model checking process, the properties of its analogy set should be paid more attention and correspondingly checked against the system implementations. On the other hand, taking the cause-effect

Table 1: The analogy set specifications for five property patterns.

Pattern name	Property specification	Analogy set specification	Explanations
Absence	LTL: $G(\neg p)$ CTL: $AG(\neg p)$	LTL: $F(p)$ CTL: $EF(p)$	OS: action p never occurs AS: action p occurs definitely
Existence	LTL: $F(p)$ CTL: $AF(p)$	LTL: $G(\neg p)$ CTL: $AG(\neg p)$	OS: action p occurs definitely AS: action p never occurs
Universality	LTL: $G(p)$ CTL: $AG(p)$	LTL: $F(\neg p)$ CTL: $EF(\neg p)$	OS: action p occurs all through AS: action p does not occur definitely
Precedence	LTL: $F(q) \rightarrow (\neg q \cup p)$ CTL: $\neg E(\neg p \cup (q \wedge (\neg p)))$	LTL: $F(q) \rightarrow (\neg q \cup \neg p)$ $G(p \rightarrow G(\neg q))$ CTL: $\neg E(p \cup (q \wedge p))$ $AG(p \rightarrow AG(\neg q))$	OS: the occurrence of action q is enabled by the occurrence of action p AS: action q occurs without the preoccurrence of action p or action p occurs without the postoccurrence of action q
Response	LTL: $G(p \rightarrow F(q))$ CTL: $AG(p \rightarrow AF(q))$	LTL: $G(p \rightarrow G(\neg q))$ $F(q) \rightarrow (\neg q \cup \neg p)$ CTL: $AG(p \rightarrow AG(\neg q))$ $\neg E(p \cup (q \wedge p))$	OS: the occurrence of action p must be followed by the occurrence of action q AS: action p occurs without the postoccurrence of action q or action q occurs without the preoccurrence of action p

relation into account, the absence, existence, and universality properties could be classified as a group, while the precedence and response properties as another group. So, if one specific property is verified in the original model checking process, the other kind of properties and its analogy set properties should be also considered to have a check against the system implementations. In this way, we complement the incompleteness of key security properties for the software model checking, and more importantly, we have more opportunities to find out the unnecessary implementation errors or the neglected implementation errors.

4.3. Refining the Models with Counterexamples. According to the framework of our LIED method in Figure 4, the analogy set of the original key properties for a specific software system is generated as a new set of properties, and then we apply the model checking, against this new set of properties, onto the system implementations directly. If all of the properties are verified successfully, we could determine that the system implementation works correctly with respect to the system specification, and then we could perform traditional conformance testing as usual. However, the LIED method is actually more willing to get a counterexample, which may reveal an existing latent implementation error. As autoproduced counterexamples could intuitively present the scenarios about how the latent errors occur, they are quite helpful for revising the system models and fixing the system implementations.

First, we present the formal definition of the counterexample from the LTS point of view, and it could be concretized with corresponding syntax towards different model checkers.

Definition 6 (unified counterexample (UCE)). A Unified Counterexample is a kind of Trace of an IOLTS model M:

$$\mathrm{UCE}(M) =_{\mathrm{def}} \{\sigma \in \mathrm{Traces}(M) \mid |\sigma| < n, n \in N\}. \quad (1)$$

Intuitively speaking, a counterexample is a specific execution of a software system, that is, a trace of detailed behaviors. The model M in the above definition refers to the system specification in program level; for example, in the Copper model checker, it is the program specification file (*.pp). According to the preceding Definition 2, a specific counterexample may be a sequence of input actions, output actions and internal actions, where internal actions reflect the value variation of corresponding program variables without external behaviors.

Based on the counterexample and the software model, a Counterexample-Guided Synchronous Simulation (CGSS) algorithm is proposed as Algorithm 1 to check whether the system model has the same behavior trace as the counterexample from the input/output point of view. If the simulation produces an empty set of synchronous traces, it means that the exception behaviors which are represented by this counterexample are not totally considered in the system model, so a UIE is actually detected. Otherwise, a nonempty set of synchronous traces reveals that the system model does have corresponding parts to deal with such exception behaviors, and an NIE is detected consequently.

If the unnecessary implementation errors are detected, the system model should be improved by adding additional parts to deal with the UIE errors that demonstrated by the counterexamples, and, the system implementations should be fixed by taking out extra program codes. Similarly, if the neglected implementation errors are detected, the system model should be modified against corresponding parts to handle the NIE errors that demonstrated by the counterexamples, and, the system implementation should be fixed by revising the inaccurate program codes.

4.4. Case Study: An AVM System. To elaborate the feasibility and effectiveness of our LIED method, an AVM software system is analyzed using this method as a representative in this section. The Copper model checker is adopted. The

Given **MS** for a system model and **ct** for a counterexample trace, where:
(1) $\exists \sigma_S \in$ Traces (MS), that is, $\exists q_0 \xrightarrow{t_0} q_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} q_n : q_i \in Q$ (MS) and $t_i \in L$ (MS), $(0 \le i \le n)$;
(2) ct could be specified as: $\exists c_0 \xrightarrow{a_0} c_1 \xrightarrow{a_1} \cdots \xrightarrow{a_m} c_m : c_j \in Q$ (ct) and $a_j \in L$ (ct), $(0 \le j \le m)$;
(3) Q (MS) $\cap$ Q (ct) = Φ and L (MS) $\supseteq$ L (ct) and $q_0 = c_0$;
Procedure CGSS {
/∗ omit all internal actions in the counterexample trace ∗/
ct_new = **Trim** (ct), where L (ct_new) $\subseteq$ LI (ct) $\cup$ LU (ct)
/∗ based on ct _new trace to perform synchronous simulation with MS model ∗/
for each pair $a_j \in L$ (ct_new) starting from a_0 **and** $t_i \in L$ (MS) starting from t_0 {
/∗ the external trace of counterexample acts the same as the trace in MS, exit to report non-empty set ∗/
if a_j is the last action of ct_new **and** $a_j = t_i$ then return non-empty set; exit;
/∗ omit considering internal actions and go for the next iteration ∗/
if $t_i \in$ LE (MS) **then** $t_i = t_{i+1}$; continue;
/∗ the same external action leads to one synchronous simulation step, and go for the next iteration ∗/
if $a_j = t_i$ **then** $a_j = a_{j+1}$, $t_i = t_{i+1}$; continue;
/∗ the different external action leads to failure simulation, and exit to report empty set ∗/
if $a_j \neq t_i$ **then** return empty set; exit;
} // end of for each pair
} // end of procedure

Algorithm 1: Counterexample-Guided Synchronous Simulation (CGSS).

IOLTS model for this AVM system is presented in Figure 2(a). Besides, we implement a program for such AVM system, which may release milk without inserting coin, just like the scenario in Figure 2(c). The core segment of this program is if (strcmp(input,"coin")==0) output_coffee (); else output_milk ();. Obviously, when something else (not a coin) is inserted, a bottle of milk is then released, and no error message is posted as expected.

Step 1 (enumerating the key properties). In traditional model checking phase, we consider a requirement that if a coin is inserted, the milk or coffee is released. We formulate this requirement into an LTL property with precedence format. Besides, its analogy set is generated, which checks the scenario that the milk or coffee is released without inserting a coin. Original property is formulated as $F(q) \rightarrow (\neg q \cup p)$. Analogy property is formulated as: $F(q) \rightarrow (\neg q \cup \neg p)$, where q stands for releasing action (output: !t or !m) and p stands for inserting action (input: ?c).

Copper supports temporal logic claims expressed in State/Event Linear Temporal Logic (SE-LTL). The syntax of SE-LTL is similar to that of LTL, except that the atomic formulas are either actions or expressions involving program variables. Therefore, the analogy property could be formulated as follows:

$$\text{ltl ExamProp } \{\#F\,(\text{output} \Longrightarrow ((!\text{output})\,\#U(!\,[\text{input} == \text{coin}]))) ;\}, \quad (2)$$

where output represents the output action output_coffee() or output_milk() in the AVM programs.

Step 2 (model checking the program). The program for the AVM system is processed into the AVM.pp file and the above property is specified into the AVM.spec file. Then, the model checking towards the program is executed using the following command.

copper - -default - -specification ExamProp AVM.pp AVM.spec - -ltl

The result of this LTL model checking is "conformance relation does not exist !! specification ExamProp is invalid. . .." Besides, a counterexample is produced correspondingly. As follows, a detailed UCE trace σ' is generated from the program variables assignment parts and the action parts in that counterexample.

σ' = ($P0$:epsilon[input = null], $P0$:epsilon[input = "Key"], strcmp, $P0$:epsilon [branch(0)], output_milk].

In this trace, $P0$:epsilon stands for internal actions that present value assignments of variables or decision of branch statements. This trace reveals that when the input is assigned with value key, not the expected value coin, the output action output_milk still occurred. The ExamProp property cannot hold against the AVM program.

Step 3 (revamping the model and program with the counterexample). We put above UCE trace σ' and the system model shown in Figure 2(a) as inputs into the CGSS algorithm. After the counterexample-guided synchronous simulation procedure, it produces an empty set. So, a UIE is actually detected. That is, a bottle of milk will be released when incorrect input is inserted, and the model has a lack of specification to deal with such error. Therefore, we improve the system model-by adding additional parts to deal with this UIE error, as shown in Figure 5. If certain incorrect input (not a coin) is inserted, the AVM system will output error messages and terminate its execution in stop state. Furthermore, we also fix the core segment of system program into "if (strcmp(input, "coin")==0) output_coffee();

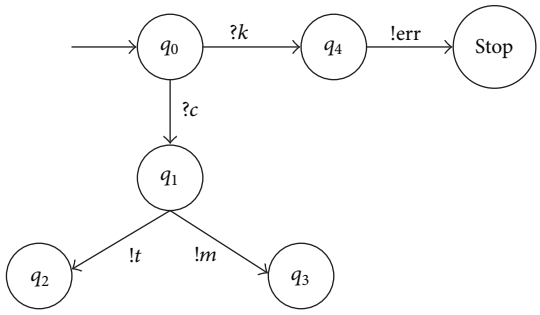

Figure 5: The improved IOLTS models for the AVM software.

else output_error();", so that the UIE error mentioned above will not occur.

Through the above exemplified execution of our LIED method towards the AVM system, its feasibility and effectiveness are demonstrated. That is, certain latent implementation errors are detected, and more importantly, the system models and implementations are well improved.

5. Conclusion

To validate the functional accuracy for a software system only applying model checking followed by conformance testing may not detect some latent implementation errors, that is, the unnecessary implementation errors and the neglected implementation errors. In this paper, the LIED method is proposed to detect such latent implementation errors proactively. Based on the analogy set of key properties, the LIED method applies model checking directly into the actual software implementation to check whether some latent implementation errors exist and utilize the counterexamples, which illustrate the exception behavior executions as intuitive guidance to improve the system models and system implementations respectively.

The LIED method is essentially well compatible with the traditional model checking and model-based conformance testing procedures. It could be applied as an effective complementary method for detecting latent implementation errors, but not for replacing the traditional model checking and conformance testing. The major advantages of our LIED method could be concluded from two aspects. First, it efficaciously complements the incompleteness of the key security properties for the software validation process. Second, it helps to construct more accurate system formal models to promote the effectiveness of model checking and model-based conformance testing, that is, the original model checking could be performed more completely because the key properties and the system models are both improved, and conformance testing could be performed more precisely because the improvement of the system models result in generating test cases with higher accuracy and stronger capability of detecting the implementation errors. In a word, the LIED method tends to be a well accelerant for better model checking and conformance testing iterative executions, where the "Under Verified" scenario is improved as expected, and consequently, these software validation methods work collaboratively to make software design and implementation more effective and more efficient. In the future, the LIED method will try to work in more complex and practical systems [23–25].

Funding

This work was supported by the National Natural Science Foundation of China (61262082, 61262017, 61163011, and 60973147), the Key Project of Chinese Ministry of Education (212025), the Inner Mongolia Science Foundation for Distinguished Young Scholars (2012JQ03), the Introduction Foundation for High-Level Talents of Inner Mongolia University, the Doctoral Fund of Ministry of Education of China [20090009110006], the Natural Science Foundation of Guangxi (2011GXNSFA018154 and 2012GXNSFGA060003), the Science and Technology Foundation of Guangxi (10169-1), and the Guangxi Scientific Research Project (201012MS274).

References

[1] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: practice and experience," *ACM Computing Surveys*, vol. 41, no. 4, article 19, pp. 1–36, 2009.

[2] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys*, vol. 41, no. 4, article 21, pp. 1–54, 2009.

[3] R. M. Hierons, K. Bogdanov, J. P. Bowen et al., "Using formal specification to support testing," *ACM Computing Surveys*, vol. 41, article 9, pp. 1–76, 2009.

[4] C. Constant, T. Jéron, H. Marchand, and V. Rusu, "Integrating formal verification and conformance testing for reactive systems," *IEEE Transactions on Software Engineering*, vol. 33, no. 8, pp. 558–574, 2007.

[5] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing Verification and Reliability*, vol. 19, no. 3, pp. 215–261, 2009.

[6] D. Peled, M. Y. Vardi, and M. Yannakakis, "Black box checking," *Journal of Automata, Languages and Combinatorics*, vol. 7, no. 2, pp. 225–246, 2002.

[7] A. Groce, D. Peled, and M. Yannakakis, "Adaptive model checking," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02)*, pp. 357–370, Springer, Grenoble, France, April 2002.

[8] J. Tretmans, "Model based testing with labelled transition systems," in *Formal Methods and Testing*, R. M. Hierons, J. P. Bowen, and M. Harman, Eds., pp. 1–38, Springer, Berlin, germany, 2008.

[9] J. Tretmans, "Test generation with inputs, outputs and repetitive quiescence," *Software-Concepts and Tools*, vol. 17, no. 3, pp. 103–120, 1996.

[10] J. Tretmans and E. Brinksma, "TorX: automated model based testing," in *Proceedings of the 1st European Conference on Model-Driven Software Engineering (ECMDSE '03)*, pp. 1–13, AGEDIS, Nuremberg, Germany, December 2003.

[11] C. Jard and T. Jéron, "TGV: theory, principles and algorithms. A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems," *International Journal*

on Software Tools for Technology Transfer, vol. 7, no. 4, pp. 297–315, 2005.

[12] V. Rusu, H. Marchand, V. Tschaen, T. Jeron, and B. Jeannet, "From safety verification to safety testing," in *Proceedings of the Testing of Communicating Systems (TestCom '04)*, pp. 160–176, Springer, Oxford, UK, March 2004.

[13] B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva, "Symbolic test selection based on approximate analysis," in *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '05)*, pp. 349–364, April 2005.

[14] M. Oostdijk, V. Rusu, J. Tretmans, R. G. De Vries, and T. A. C. Willemse, "Integrating verification, testing, and learning for cryptographic protocols," *Lecture Notes in Computer Science*, vol. 4591, pp. 538–557, 2007.

[15] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *ACM SIGSOFT Software Engineering Notes*, vol. 24, pp. 146–162, 1999.

[16] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural, "A temporal logic based theory of test coverage and generation," in *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02)*, pp. 327–341, Springer, Grenoble, France, April 2002.

[17] D. A. da Silva and P. D. L. Machado, "Towards test purpose generation from CTL properties for reactive systems," *Electronic Notes in Theoretical Computer Science*, vol. 164, no. 4, pp. 29–40, 2006.

[18] G. Fraser and A. Gargantini, "An evaluation of model checkers for specification based test case generation," in *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation (ICST '09)*, pp. 41–50, IEEE Press, Denver, Colo, USA, April 2009.

[19] Software Engineering Institute, CMU, *Copper Manual, Tutorial, and Specification Grammar*, http://www.sei.cmu.edu/library/abstracts/whitepapers/copper.cfm

[20] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the International Conference on Software Engineering (ICSE '99)*, pp. 411–420, IEEE Press, Los Angeles, Calif, USA, May 1999.

[21] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, *Spec Patterns*, The Santos Laboratory, Kansas State University.

[22] A. Fedeli, F. Fummi, and G. Pravadelli, "Properties incompleteness evaluation by functional verification," *IEEE Transactions on Computers*, vol. 56, no. 4, pp. 528–544, 2007.

[23] J. B. Wang, M. Chen, X. Wan, and C. Wei, "Ant-colony-optimization-based scheduling algorithm for uplink CDMA nonreal-time data," *IEEE Transactions on Vehicular Technology*, vol. 58, no. 1, pp. 231–241, 2009.

[24] J. B. Wang, H. M. Chen, M. Chen, and J. Z. Wang, "Cross-layer packet scheduling for downlink multiuser OFDM systems," *Science in China F*, vol. 52, no. 12, pp. 2369–2377, 2009.

[25] J.-B. Wang, Y. Jiao, X. Song, and M. Chen, "Optimal training sequences for indoor wireless optical communications," *Journal of Optics*, vol. 14, no. 1, Article ID 015401, 2012.