Program Repair Suggestions from Graphical State-Transition Specifications

Farn Wang[1,2] and Chih-Hong Cheng[1]

[1] Dept. of Electrical Engineering, National Taiwan University, Taiwan, ROC
[2] Grad. Inst. of Electronic Engineering, National Taiwan University, Taiwan, ROC
farn@cc.ee.ntu.edu.tw

Abstract. In software engineering, graphical formalisms, like state-transition tables and automata, are very often indispensable parts of the specifications. Such a formalism usually leads to specification refinement that maintains the simulation/bisimulation relation between an implementation and a specification. We investigate how to use formal techniques to generate suggestions for repairing a program that breaks the bisimulation relation with a graphical specification. We use state graphs as a unified representation of the program models and specifications. We propose a technique that may evaluate the cost of a repair. We present a PTIME heuristic algorithm that suggests how to repair a model state graph. We then explain how to derive repair suggestions for programs from the repair for state graphs. Finally, we report our experiment that checks the performance of our repair algorithms and the costs of our repairs.

Keywords: state graph, state transition relation, repair, graph theory, cost, evaluation, equivalence, bisimulation.

1 Introduction

The construction of large complex software with quality assurance is becoming more important than ever. In general, quality assurance is achieved with verification techniques, i.e., checking if the behavior of a design meets a specification. Up to now, for program verification, various techniques have been developed, including testing [14] and model checking [5]. Once a bug is reported in the verification process, locating and repairing the bug still rely heavily on human intervention which is costly, time-consuming, and error-prone. In fact, the process of program repair remains to be the least automated in system development. When talking about repairing, the cost is usually taken into account. Thus, without taking repair cost into consideration, research work in program repair is not likely to be useful in practice. This work is to develop techniques for repair suggestions of programs with a cost concept against graphical state-transition specifications.

Graphical specification formalisms have been widely used in software engineering and telecommunication industry. Examples are the state-transition diagrams used in the specification of many protocols, the statecharts of UML, the

K. Suzuki et al. (Eds.): FORTE 2008, LNCS 5048, pp. 185–200, 2008.

abstract machines of SDL, automata, ..., etc. In this work, we adopt such a formalism, called *state graph*, as a unified representation for both program models and state-transition specifications. There are many algorithms and tools that can construct the state graphs of programs automatically [1].

There are many definitions for the verification between two state graphs. For example, we can compare sets of traces of the two state graphs. However, state graphs are usually used as a suggestion for the behavior structures of a program in a state-by-state and transition-by-transition way. Thus we feel that *simulation checking* between state graphs is a better choice in this work. Intuitively, one state graph A_m *is simulated by* another A_s if and only if every transition that A_m can make can also be matched by A_s at a corresponding state. But this framework sometimes is still not good enough for practical verification in the industry. For one thing, a specification state graph could be vacuously satisfied by a faulty program that yields no behavior at all. One way to cope with this problem is to also specify some good behaviors which the program must exhibit. Specifically, we can have a pair of state graphs, $A_s^{(l)}$ and $A_s^{(u)}$ respectively for the lower-bound and the upper-bound specifications. Given the model state graph A_m of a program, we can thus verify whether $A_s^{(1)}$ is simulated by A_m and A_m is simulated by $A_s^{(u)}$.

However, we feel that the framework of simulation-checking with both a lower-bound and an upper-bound specifications is a little complicated and may blur the technical presentation in this article. Instead, we use a less involved framework called *bisimulation checking* [13,15]. Intuitively, two state graphs are *bisimulation equivalent* if and only if for every corresponding state pairs of the two graphs, every transition that the one graph can make at a state can also be matched by the other graph at a corresponding state, and vice versa. In a not very rigorous sense, bisimulation-checking is like simulation-checking when the lower-bound and upper-bound state-transition specifications are the same. The techniques we present in this work for the framework of bisimulation-checking should also be applicable to the framework of simulation-checking with lower-bound and upper-bound specifications.

In repairing a program for a specification, engineers usually can evaluate whether a repair is better than another. For example, a better repair might introduce less changes to a program, might run more efficiently, might use less memory, might be more readable, ..., and etc. It is easy to see that there are many dimensions in evaluating how good a repair decision is. Thus it is in general difficult to define a formal approach to evaluate repairs in a way that matches human engineers' intuition. Anyway, we still feel it is important to have the first step in formalizing the evaluation of repairs. In this work, we borrow the graph edit-distance concept in graph theory for the evaluation of the '*cost*' of repairs. A *repair* is defined as a sequence of *edit operations* to A_m to make A_m and A_s bisimulation equivalent. We consider several types of *edit operations* to state graphs. The length of an edit operation sequence naturally defines the *cost* of the corresponding repair. In such a context, cost of repairs can help engineers to evaluate the degree of changes to be introduced with a repair. This can be

useful in maintaining legacy software when engineers may prefer not to introduce significant changes.

In figure 1, we present our framework of verification and program repair suggestion. We construct the state graphs from a program and a graphical specification of a state transition relation. We then check the bisimulation equivalence

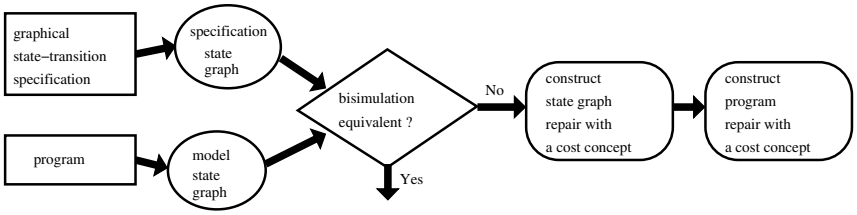

Fig. 1. Framwork of verification and program repair

between the program state graph and the specification state graph. If they are not bisimulation equivalent, then we use the techniques in this work to construct suggestions for repairing the program with a cost concept.

In this work, we establish an upper-bound on the cost to repair a model state graph with respect to a specification one. We also present a logic-based algorithm for the calculation of an upper-bound for the minimum repair cost. We then present a PTIME heuristic algorithm for constructing repairs. We have implemented the heuristic algorithm and compared its performance with a straightforward exploration procedure that searches through the space of repairs. We have experimented against several benchmarks. Our heuristic repair algorithm can sometimes find a repair at a cost lower than the just-mentioned upper-bound. We feel that the heuristic algorithm could be used as a foundation for further investigation in this research direction. Finally, we explain how to convert the repair for state graphs to the repair suggestions for programs.

The rest of the paper is presented as follows. Section 2 reviews related work. Section 3 briefly defines state graphs and bisimulation and explains how programs can be converted to models as state graphs. Section 4 discusses the cost evaluation of repairs. Section 5 establishes an upper-bound of the minimum-cost repair for a given repair task and presents an algorithm for calculating the upper-bound. Section 6 presents algorithms for the construction of repairs with a cost concept and explains how to repair a program based on the repair of the corresponding model state graph. Section 7 reports our experiment. Section 8 is for the conclusion and possible future directions.

2 Related Work

Jobstmann et al. viewed the program repair problem as a game. Given a set of suspicious statements (information from fault localization), they first relax the

constraints on those suspicious statements and then look for a further constraint of the statements to make the program satisfy specifications [12]. Thus the possible program repairs are restricted to the original architectures of the models. The work of Griesmayer et al. could be viewed as an extension in this direction [9]. In contrast, our framework does not constrain ourselves to those repairs conforming to the original architectures of the model automata. We allow for any repaired model that can be represented as a state graph. Our framework also enables the analysis of repair costs. Moreover, our framework does not rely on the availability of a fault localizer.

There have been discussions in the Artificial Intelligence (AI) community on repair automation. We discuss two of them in the following. Buccafurri et al. argued, with examples, for the connection between the system repair problem and abductive theory revision problem [3]. They also argued that the repair cost can be estimated with the length of the corresponding edit operation sequence and proposed heuristics to avoid redundancy and optimize in the search of the minimum repair.

Ding and Zhang defined the basic repair steps of Kripke model for specifications in *LTL (linear-time temporal logic)* [6]. To formalize the concept of repair cost, they defined the ordering among repairs and presented theorems in characterizing the minimum repairs for specifications like $F\psi$ and $\psi_1 \wedge \psi_2$. They also presented an algorithm to repair Kripke models for CTL specifications [7].

In this work, we also present a logic-based algorithm for the calculation of MCS between graphs. At this moment, there are many tools that can construct the MCS between two graphs, for instance, *SimPack* [16]. But, to our knowledge, no existing tools support the construction of MCS' of graphs with both arc and vertex labels.

3 State Graphs

For convenience, we have the following notations. Given a set or a sequence V, the size (number of elements) of V is denoted $|V|$. Given a function f, we let f^{-1} be the inverse of f. Also, 'iff' is a shorthand for "if and only if."

Definition 1. (State graphs) A *state graph* A on a set P of atomic propositions and an alphabet Σ is a tuple (Q, P, μ, Σ, E) with the following constraints.

- Q is a finite set of *states*.
- P is a finite set of *atomic propositions*. We assume there is an atomic proposition $ini \in P$ that denotes whether a state is initial.
- $\mu : Q \mapsto (P \mapsto \{false, true\})$ is a labeling function for the states.
- Σ is a finite set of input symbols.
- $E \subseteq (Q \times \Sigma \times Q)$ is a finite set of *transitions*.

Also, we let $ini(A) = \{q \mid \mu(q, ini)\}$ be the set of initial states of A. ■

There are many known techniques that allow us to abstract a program into a state graph [1]. Thus state graph can be used as a unified representation for both our program models and our graphical state-transition specifications. For conciseness

of presentation, in this work, we use the following *bisimulation relation* [13] to define the verification problem between two state graphs.

Definition 2. (Bisimulation of state graphs) Given a set P of atomic propositions, a set Σ of input symbols, and two state graphs $A_1 = (Q_1, P, \mu_1, \Sigma, E_1)$ and $A_2 = (Q_2, P, \mu_2, \Sigma, E_2)$, a *bisimulation* B between A_1 and A_2 is a relation $B \subseteq Q_1 \times Q_2$ such that for every $(q_1, q_2) \in B$, the following restrictions hold.

- $\mu_1(q_1) = \mu_2(q_2)$.
- For every $(q_1, a, q_1') \in E_1$, there is a $(q_1', q_2') \in B$ with $(q_2, a, q_2') \in E_2$.
- For every $(q_2, a, q_2') \in E_2$, there is a $(q_1', q_2') \in B$ with $(q_1, a, q_1') \in E_1$.

A_1 and A_2 are *bisimulation equivalent*, in symbols $A_1 \equiv A_2$, iff there is a bisimulation B between A_1 and A_2 with the following restrictions.

- For every $q_1 \in \mathit{ini}(A_1)$, there is a $q_2 \in \mathit{ini}(A_2)$ with $(q_1, q_2) \in B$.
- For every $q_2 \in \mathit{ini}(A_2)$, there is a $q_1 \in \mathit{ini}(A_1)$ with $(q_1, q_2) \in B$. ■

Bisimulation preserves all properties expressible in the propositional μ-calculus, which subsumes CTL* [8] in expressiveness. The maximal bisimulation between two state graphs can be constructed in deterministic polynomial time [15].

4 Repairs and Their Cost Estimation

As we have said that, there are good repairs and bad repairs. It is in general difficult to evaluate how good a repair is. We first formalize the concept of repairs to state graphs. As in [3,6,7], we may define a repair of a model state graph as a sequence of graph-edit operations that transforms the graph to one that is bisimulation equivalent to a specification. In a repair, we allow the following four types of basic edit operations. Suppose we are given a state graph $A = (Q, P, \mu, \Sigma, E)$.

- **State addition**: Given a state q and a set $L \subseteq P$, $\lambda X.\mathit{state\_add}(X, q, L)$ is an operation that adds state q to X with labels in L. Formally speaking, $\mathit{state\_add}(A, q, L)$ is a new state graph $(Q \cup \{q\}, P, \mu', \Sigma, E)$ such that μ' is identical to μ except that $\mu'(q) = L$. Note that if $q \in Q$, then the addition has no effect.
- **State deletion**: Given a state q, $\lambda X.\mathit{state\_del}(X, q)$ is an operation that takes state q out of state graph X. Formally speaking, $\mathit{state\_del}(A, q)$ is a new state graph $(Q - \{q\}, P, \mu, \Sigma, E)$. Note that if $q \notin Q$, then the operation does not have an effect. Also deleting a state with incoming or outgoing transitions has no effect.
- **Transition addition**: Given two states $q, q' \in Q$ and an $a \in \Sigma$, $\lambda X.\mathit{xtion\_add}(X, q, a, q')$ is an operation that adds transition (q, a, q') to state graph X. Formally speaking, $\mathit{xtion\_add}(A, q, a, q')$ is a new state graph $(Q \cup \{q\}, P, \mu, \Sigma, E \cup \{(q, a, q')\})$. In case $q \notin Q$, $q' \notin Q$, or $a \notin \Sigma$, $\mathit{xtion\_add}(A, q, a, q') = A$.
- **Transition deletion**: Given two states $q, q' \in Q$ and an $a \in \Sigma$, $\lambda X.\mathit{xtion\_del}(X, q, a, q')$ is an operation that takes transition (q, a, q') out of state graph X. Formally, $\mathit{xtion\_del}(A, q, a, q')$ is a new state graph

$(Q \cup \{q\}, P, \mu, \Sigma, E - \{(q, a, q')\})$. In case $q \notin Q$ or $q' \notin Q$, or $a \notin \Sigma$, $\mathit{xtion\_del}(A, q, a, q') = A$.

An *edit sequence* is a sequence of edit operations. Given an edit sequence $e_1 e_2 \dots e_n$ on a state graph A, the result of the sequence on A, in symbols $Ae_1 e_2 \dots e_n$, is defined inductively as follows.

- $A\epsilon = A$ where ϵ is the null sequence.
- $A\,(\lambda X.\mathit{state\_add}(X, q, L))\, e_2 \dots e_n = \mathit{state\_add}(A, q, L) e_2 \dots e_n$.
- $A\,(\lambda X.\mathit{state\_del}(X, q))\, e_2 \dots e_n = \mathit{state\_del}(A, q) e_2 \dots e_n$.
- $A\,(\lambda X.\mathit{xtion\_add}(X, q, a, q'))\, e_2 \dots e_n = \mathit{xtion\_add}(A, q, a, q') e_2 \dots e_n$.
- $A\,(\lambda X.\mathit{xtion\_del}(X, q, a, q'))\, e_2 \dots e_n = \mathit{xtion\_del}(A, q, a, q') e_2 \dots e_n$.

The *cost* of a repair $\sigma = e_1 \dots e_n$ is defined as $|\sigma| = n$, i.e., the length of σ. For example, in figure 2, we have (a) for a model graph and (b) for a specification graph. The initial states are with incoming arrows without a source. (c) is the obtained from a repair of (a) for (b) with the minimum repair cost two. A repair is the following edit sequence.

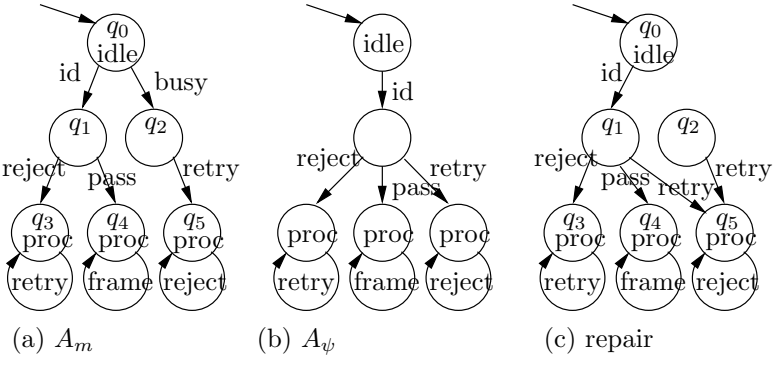

Fig. 2. An example of repair

$$(\lambda X.\mathit{xtion\_del}(q_0, \text{busy}, q_2))(\lambda X.\mathit{xtion\_add}(q_1, \text{retry}, q_5))$$

5 Upper-Bounds for Minimum Repair Cost

State graphs are in fact directed graphs with states and arc labels. In this section, we base on graph theory, specifically the work of Bunke [4], to derive an upper-bound on minimum repair cost for bisimulation equivalence.

5.1 Upper-Bounds from the Graph Theory

We can define the *isomorphism* between state graphs. Two state graphs $A_1 = (Q_1, P, \mu_1, \Sigma, E_1)$ and $A_2 = (Q_2, P, \mu_2, \Sigma, E_2)$ are *isomorphic* if there is a bijective function β from Q_1 to Q_2 such that

- for all $q \in Q_1$, $\mu_1(q) = \mu_2(\beta(q))$;
- for all $(q_1, a, q_2) \in E_1$, $(\beta(q_1), a, \beta(q_2)) \in E_2$;
- for all $(q_1, a, q_2) \in E_2$, $(\beta^{-1}(q_1), a, \beta^{-1}(q_2)) \in E_1$.

We have the following intuitive lemma.

Lemma 1. *Given a model graph A_m, a specification graph A_s, and an edit sequence σ, if $A_m\sigma$ is isomorphic to A_s, then σ is a repair.*

Proof : True since isomorphic state graphs are bisimulation equivalent. ■

Lemma 1 suggests that we can use the length of the shortest edit sequence that changes A_m to A_s as an upper-bound for the minimum repair cost. The upper-bound can be used to bound our exploration in the search for a minimum repair from A_m to A_s.

In the following, since we may use graph theory to handle state graphs, sometimes we conveniently use the terms in graph theory to call the equivalent structures in our state graphs. For example, we may also call a state a *vertex* and a transition an *arc*. The *size* of a graph $A = (Q, P, \mu, \Sigma, E)$, denoted $|A|$, is defined as $|Q| + |E|$. Given a state q, we may write $q \in A$ iff $q \in Q$. Also given a transition (q, a, q'), we may write $(q, a, q') \in A$ iff $(q, a, q') \in E$. A *subgraph* $A' = (Q', P, \mu, \Sigma, E')$ of a state graph $A = (Q, P, \mu, \Sigma, E)$ is a graph such that $Q' \subseteq Q$ and $E' \subseteq E$. Note that we let A and A' share the same state-labeling function for the simplicity of presentation.

Definition 3. (Maximum common subgraph) Let A_1 and A_2 be two graphs and A_1' and A_2' be subgraphs of A_1 and A_2 respectively. We call A_1' (or A_2') a *common subgraph* of A_1 and A_2 if A_1' and A_2' are isomorphic. A graph G is a *maximum common subgraph (MCS)* of A_1 and A_2 if G is a common subgraph of A_1 and A_2 and for all common subgraphs G' of A_1 and A_2, $|G'| \leq |G|$. ■

The relation between edit sequences and MCS was first presented by Bunke in [4]. Bunke's work is based on the assumption that the size of a graph is only relevant to the number of vertices. Moreover, the edit operations of arcs in his work are all free. In contrast, we assume that the cost of an edit operation to an arc (transition) is also one. We have adapted the following lemma from [4] for the relation between edit sequences and MCS.

Lemma 2. *Suppose we are given three state graphs A_1, A_2, and A_c such that A_c is an MCS of A_1 and A_2. Then the shortest edit sequence that changes A_1 to A_2 is of length $|A_1| + |A_2| - 2|A_c|$.* ■

Due to page-limit, we have left the proof to a full version of the paper in our tool website. With lemmas 1 and 2, we can establish the following lemma.

Lemma 3. *Suppose we are given a model state graph A_m and a specification state graph A_s. If A_c is an MCS of A_m and A_s, then the minimum repair cost of A_m for A_s is no greater than $|A_m| + |A_s| - 2|A_c|$.* ■

Due to page-limit, we have left the proof of the lemma to a full version of the paper in our tool website. We use figure 3 to explain lemma 3. The parts circled

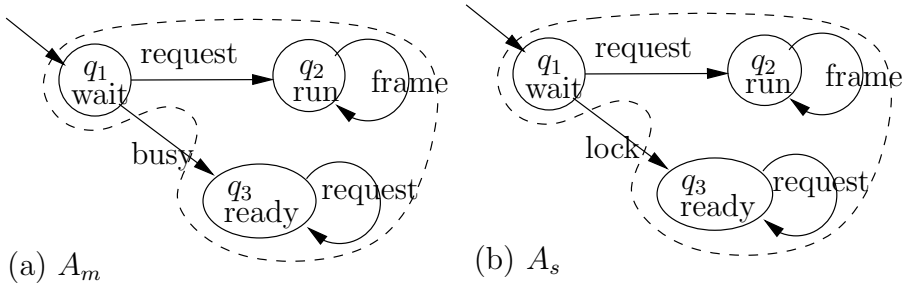

Fig. 3. Two state graphs

with dashed lines are the MCS, say A_c, of the two state graphs. The minimum repair cost is no greater than $|A_m| + |A_s| - 2|A_c| = 7 + 7 - 2 \times 6 = 2$.

The following lemma shows that the upper-bound established with lemma 3 is actually tight. We can establish the faimily of A_m^i's and A_s^i's in figure 4 that share no MCS.

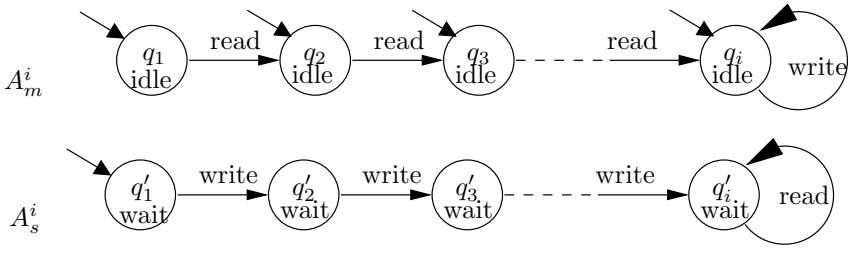

Fig. 4. A family of A_m^i and A_s^i with tight upper-bound repair cost

Lemma 4. *For the family of state graphs in figure 4, for each positive integer i, the minimum cost of repair of A_m^i for A_s^i is $|A_m^i| + |A_s^i|$.*

Proof : As can be seen from figure 4, for each i, there is no common subgraph between A_m^i and A_s^i. Moreover, if any state in A_m^i remains to be initial, A_m^i cannot be repaired to be bisimulation equivalent with A_s^i. To remove states in A_m^i, we first have to remove all transitions in A_m^i. This costs $|A_m^i|$ edit operations. Then we need $|A_s^i|$ edit operations to add A_s^i to A_m^i. In this way, the repaired model becomes isomorphic to A_s^i. According to lemma 1, the repaired model is thus bisimulation equivalent to A_s^i. The cost is thus $|A_m^i| + |A_s^i|$ for each i. ■

5.2 A Logic-Based Algorithm for the MCS

Our algorithm is built on an MCS construction algorithm. Note that the calculation of MCS is an NP-complete problem [10]. Our motivation is that with

proper encoding of the logic formulas in advanced data-structures, like BDD [2], we have a better chance to calculate MCS efficiently in the average cases. Specifically, we want to construct a logic formula that characterizes the common subgraphs between two state graphs. A *solution* (satisfying truth assignments) to the formula can be used to help us constructing a common subgraph. An MCS then corresponds to a *maximal* solution that assigns the most number of 1's to the variables.

Given a set V of Boolean variables, a *formula* η of V can be inductively constructed with rule "$\eta ::= v \mid \neg\eta_1 \mid \eta_1 \vee \eta_2$." Standard shorthands like $\eta_1 \wedge \eta_2$, $\eta_1 \rightarrow \eta_2$, and $\eta_1 \leftrightarrow \eta_2$ are also allowed in this work. A *truth value* is either *true* or *false*. An *interpretation* of a formula is a mapping from its set of Boolean variables to truth values. An interpretation I *satisfies* a formula η, in symbols $I \models \eta$, if the following inductive conditions are maintained.

- $I \models v$ iff $I(v) = \mathit{true}$.
- $I \models \neg\eta_1$ iff it is not the case that $I \models \eta_1$.
- $I \models \eta_1 \vee \eta_2$ iff either $I \models \eta_1$ or $I \models \eta_2$.

I is a *solution* to η iff $I \models \eta$. Given two solutions I and I', if for every $v \in V$, $I(v)$ implies $I'(v)$, we say I' is *no smaller than* I. A *maximal* solution is no smaller than any other solutions.

In our formulas, we use the following Boolean variables for the correspondence between states and transitions of two state graphs $A_1 = (Q_1, P, \mu_1, \Sigma, E_1)$ and $A_2 = (Q_2, P, \mu_2, \Sigma, E_2)$.

$$\left\{c_{q_2}^{q_1} \mid q_1 \in Q_1, q_2 \in Q_2\right\} \cup \left\{c_{(q_2,a,q_2')}^{(q_1,a,q_1')} \mid (q_1,a,q_1') \in E_1, (q_2,a,q_2') \in E_2\right\}.$$

Intuitively, for each $q_1 \in Q_1$ and $q_2 \in Q_2$, $c_{q_2}^{q_1}$ is true iff state q_1 corresponds to state q_2 in the MCS; for each $(q_1,a,q_1') \in E_1$ and $(q_2,a,q_2') \in E_2$, $c_{(q_2,a,q_2')}^{(q_1,a,q_1')}$ is true iff transition (q_1,a,q_1') corresponds to transition (q_2,a,q_2') in the MCS. In the following, we list the restrictions of the correspondence and their respective formulas.

- **State equivalence mutual exclusion**: A state q_1 cannot correspond to more than one state in q_2; and vice versa.
$$\mathrm{VEME}(Q_1,Q_2) \equiv \bigwedge_{q_1 \in Q_1, q_2 \in Q_2} \left(c_{q_2}^{q_1} \rightarrow \left(\begin{array}{c} \bigwedge_{\bar{q}_2 \in Q_2 - \{q_2\}} \neg c_{\bar{q}_2}^{q_1} \\ \wedge \bigwedge_{\bar{q}_1 \in Q_1 - \{q_1\}} \neg c_{q_2}^{\bar{q}_1} \end{array} \right) \right)$$
- **State equivalence structure**: Corresponding states must have the same labels.
$$\mathrm{VES}(Q_1,\mu_1,Q_2,\mu_2) \equiv \bigwedge_{q_1 \in Q_1, q_2 \in Q_2, \mu_1(q_1) \neq \mu_2(q_2)} \neg c_{q_2}^{q_1}$$
- **Transition equivalence mutual exclusion**: A transition in E_1 cannot correspond to more than one transition in E_2; and vice versa.
$$\bigwedge_{\substack{(q_1,a,q_1') \in E_1, \\ (q_2,a,q_2') \in E_2}} \left(c_{(q_2,a,q_2')}^{(q_1,a,q_1')} \rightarrow \left(\begin{array}{c} \bigwedge_{(\bar{q}_2,a,\bar{q}_2') \in Q_2 - \{(q_2,a,q_2')\}} \neg c_{(\bar{q}_2,a,\bar{q}_2')}^{(q_1,a,q_1')} \\ \wedge \bigwedge_{(\bar{q}_1,a,\bar{q}_1') \in Q_1 - \{(q_1,a,q_1')\}} \neg c_{(q_2,a,q_2')}^{(\bar{q}_1,a,\bar{q}_1')} \end{array} \right) \right)$$
- **Transition equivalence structure**: If two transitions correspond to each other, then their sources must correspond to each other, their destinations must correspond to each other, and their transition labels must be the same.

$$\mathrm{AES}(E_1, E_2) \equiv \bigwedge_{(q_1,a,q_1')\in E_1,(q_2,a,q_2')\in E_2} \left(c_{(q_2,a,q_2')}^{(q_1,a,q_1')} \rightarrow \left(c_{q_2}^{q_1} \wedge c_{q_2'}^{q_1'}\right)\right)$$

We then construct the following formula for *common subgraph restriction*: $\mathrm{CSR}(A_1, A_2)$ as the following conjunction.

$$\mathrm{VEME}(Q_1, Q_2) \wedge \mathrm{VES}(Q_1, \mu_1, Q_2, \mu_2) \wedge \mathrm{AEME}(E_1, E_2) \wedge \mathrm{AES}(E_1, E_2).$$

Given a solution I of $\mathrm{CSR}(A_1, A_2)$, we can construct the common subgraph $\mathrm{CS}(A_1, A_2, I)$ corresponding to I as follows.

$$\left(\left\{q_2 \mid \exists q_1 (I(c_{q_2}^{q_1}))\right\}, P, \mu_2, \Sigma, \left\{(q_2, a, q_2') \,\middle|\, \exists q_1 \exists a \exists q_1' \left(I\left(c_{(q_2,a,q_2')}^{(q_1,a,q_1')}\right)\right)\right\}\right)$$

Note that we use a subgraph in A_2 to represent the common subgraph. We can also do it the other way around. With the restrictions in the above, we can show that each solution of $\mathrm{CSR}(A_1, A_2)$ fully describes a common subgraph.

Lemma 5. *Given two state graphs* $A_1 = (Q_1, P, \mu_1, \Sigma, E_1)$ *and* $A_2 = (Q_2, P, \mu_2, \Sigma, E_2)$, A_c *is a common subgraph of* A_1 *and* A_2 *iff there is a solution* I *of* $CSR(A_1, A_2)$ *such that* A_c *is isomorphic to* $CS(A_1, A_2, I)$.

Proof : The correctness of the lemma can be established by checking that $\mathrm{CSR}(A_1, A_2)$ correctly encodes all the constraints for MCS construction. ■

Also a maximal solution encodes an MCS as stated with the following lemma.

Lemma 6. *Given two state graphs* $A_1 = (Q_1, P, \mu_1, \Sigma, E_1)$ *and* $A_2 = (Q_2, P, \mu_2, \Sigma, E_2)$, A_c *is an MCS of* A_1 *and* A_2 *iff there is a maximal solution* I *of* $CSR(A_1, A_2)$ *such that* A_c *is isomorphic to* $CS(A_1, A_2, I)$.

Proof : The lemma follows from lemma 5 and the fact that the number of '1's in a solution actually is equal to the size of the corresponding MCS. ■

With lemma 6, we have the following algorithm for MCS construction.

$\mathrm{MCS}(A_1, A_2)$ {
  Find a maximal solution I for $\mathrm{CSR}(A_1, A_2)$. Return $\mathrm{CS}(A_1, A_2, I)$.
}

Note that we do not elaborate on how to find the maximal solutions. In this work, we use JDD (Java BDD library) [11] to construct $\mathrm{CSR}(A_1, A_2)$. JDD can list all solutions of a formula. A maximal solution has the most number of 1's in the listing. It is also possible to take advantage of the structure-sharing capability of BDDs [2] and design a recursive procedure to efficiently search for the maximal solutions. But due to page-limit, we omit the discussion here.

6 Techniques for Repair Suggestions with a Cost Concept

In a real-world project, minimum cost repairs may be difficult and costly to construct. To improve the performance of automated repair tools, sometimes we

may have to settle for quick repairs that may not be of minimum cost. In the following, we present a PTIME heuristic algorithm for constructing repairs of model state graphs. Given a model graph A_m and a specification graph A_s, the algorithm consists of the following three steps.

- Identifying the common structure of A_m and A_s. Here we use the maximal bisimulation between A_m and A_s instead of MCS for the common structure.
- Disabling the difference from A_m to the common structure. The idea is to make all states in the difference from A_m to the common structure unreachable from any initial states.
- Gluing a compact version of the difference from A_s to the common structure to the common structure.

According to lemma 4, these steps do not save time in the worst case. However, according to the experiment, in many cases, they yield repairs with costs lower than the upper-bounds predicted by lemma 3.

At the end, we also discuss how to derive repair suggestions of programs based on the repairs of model state graphs.

6.1 Identifying the Common Structure Between A_m and A_s

Given two state graphs A_1 and A_2, there are classical algorithms that construct the maximal bisimulation, in symbols $B(A_1, A_2)$, between A_1 and A_2. Given a state graph $A = (Q, P, \mu, \Sigma, E)$, $B(A, A)$ is the maximal bisimulation between A and itself. Given a state $q \in Q$, the *bisimulation equivalence class* of q, in symbols $[q]$, is the set of states that are bisimulation equivalent to q in A with respect to $B(A, A)$. Formally speaking, $[q] = \{q' \mid q' \in Q, (q, q') \in B(A, A)\}$. The *bisimulation quotient* of a state graph $A = (Q, P, \mu, \Sigma, E)$, in symbols $[A]$, is a state graph $(\{[q] \mid q \in Q\}, P, \mu', \Sigma, \{([q], a, [q']) \mid (q, a, q') \in E\})$ such that for each $q \in Q$, $\mu'([q]) = \mu(q)$.

Suppose we have two state graphs $A_1 = (Q_1, P, \mu_1, \Sigma, E_1)$ and $A_2 = (Q_2, P, \mu_2, \Sigma, E_2)$. For each $i \in [1, 2]$, we use $\langle A_i \rangle_{B(A_1,A_2)}$ to denote the subgraph of A_i in the maximal bisimulation $B(A_1, A_2)$. That is, $\langle A_i \rangle_{B(A_1,A_2)}$ is a subgraph $(Q, P, \mu_i, \Sigma, \{(q, a, q') \mid q \in Q, q' \in Q, (q, a, q') \in E_i\})$ of A_i such that $Q = \{q \mid q \in Q_i, \exists q' \in Q_{3-i}((q, q') \in B(A_1, A_2) \vee (q', q) \in B(A_1, A_2))\}$. Given a model state graph A_m and a specification state graph A_s, we can view $\langle A_m \rangle_{B(A_m,A_s)}$ and $\langle A_s \rangle_{B(A_m,A_s)}$ as the common structure between A_m and A_s.

6.2 Identifying of the Difference Between A_m and A_s

According to the definition of bisimulation, we know that $\langle A_m \rangle_{B(A_m,A_s)}$ and $\langle A_s \rangle_{B(A_m,A_s)}$ are bisimulation equivalent. They can be viewed as intermediate products in the repair process with all 'unwanted' components removed from A_m. Assume that $A_m = (Q_m, P, \mu_m, \Sigma, E_m)$ and $A_s = (Q_s, P, \mu_s, \Sigma, E_s)$.

Assume that $\langle A_m \rangle_{B(A_m,A_s)} = (Q_b, P, \mu_m, \Sigma, E_b)$. The difference from A_m to $\langle A_m \rangle_{B(A_m,A_s)}$, in symbols $A_m - \langle A_m \rangle_{B(A_m,A_s)}$, can be straightforwardly defined as the following state graph.

$$\left(\{q \mid q \in Q_m - Q_b\}, P, \mu_m, \Sigma, \left\{([q], a, [q']) \;\middle|\; \begin{array}{l}(q, a, q') \in E_m, \\ q \in Q_m - Q_b, q' \in Q_m - Q_b\end{array}\right\}\right)$$

We need to disable the effect of $A_m - \langle A_m \rangle_{B(A_m,A_s)}$ for the repair.

Assume that $\langle A_s \rangle_{B(A_m,A_s)} = (Q_b, P, \mu_m, \Sigma, E_b)$. Similarly, we can also define $A_s - \langle A_s \rangle_{B(A_m,A_s)}$ and use it as the difference from A_s to $\langle A_s \rangle_{B(A_m,A_s)}$. However, the 'difference' could still be too big. We propose only to glue the difference from the bisimulation quotient of A_s to $\langle A_s \rangle_{B(A_m,A_s)}$. Specifically, we define this difference, in symbols $[A_s] - \langle A_s \rangle_{B(A_m,A_s)}$, as the following state graph.

$$\left(\{[q] \mid q \in Q_s - Q_b\}, P, \mu, \Sigma, \left\{([q], a, [q']) \;\middle|\; \begin{array}{l}(q, a, q') \in E_s, \\ q \in Q_s - Q_b, q' \in Q_s - Q_b\end{array}\right\}\right).$$

We require that for each $q \in Q_s - Q_b$, $\mu([q]) = \mu_s(q)$. This graph captures the behavior of those states in $Q_s - Q_b$ in the bisimulation quotient of A_s.

6.3 Constructing Repair Based on the Common Structure and the Difference

With the concepts defined in the above, we are now ready to present our PTIME algorithm for the repair of A_m for A_s. For convenience, assume that $A_m = (Q_m, P, \mu_m, \Sigma, E_m)$, $A_s = (Q_s, P, \mu_s, \Sigma, E_s)$, $\langle A_m \rangle_{B(A_m,A_s)} = (Q_b, P, \mu_m, \Sigma, E_b)$, and $[A_s] - \langle A_s \rangle_{B(A_m,A_s)} = (Q_d, P, \mu_s, \Sigma, E_d)$. Intuitively, the algorithm consists of the following two steps.

- *Disabling* $A_m - \langle A_m \rangle_{B(A_m,A_s)}$ *in* A_m. We need to delete all initial states in $A_m - \langle A_m \rangle_{B(A_m,A_s)}$. In addition, we also need to delete all transitions to and from those initial states in $A_m - \langle A_m \rangle_{B(A_m,A_s)}$.
- *Gluing* $[A_s] - \langle A_s \rangle_{B(A_m,A_s)}$ *to* $\langle A_m \rangle_{B(A_m,A_s)}$. This involves the construction of appropriate transitions between $[A_s] - \langle A_s \rangle_{B(A_m,A_s)}$ and $\langle A_m \rangle_{B(A_m,A_s)}$.

The repair generates a graph $(Q_b \cup Q_d, P, \mu, \Sigma, E)$ with the following constraints.

- $E = E_b \cup E_d$
  $\cup \{(q_1, a, [q_2]) \mid \exists (q_1, q') \in B(A_m, A_s)((q', a, q_2) \in E_s \wedge [q_2] \in Q_d)\}$
  $\cup \{([q_2], a, q_1) \mid \exists (q_1, q') \in B(A_m, A_s)((q_2, a, q') \in E_s \wedge [q_2] \in Q_d)\}$
- For each $q \in Q_b$, $\mu(q) = \mu_m(q)$. For each $[q] \in Q_d$, $\mu([q]) = \mu_s(q)$.

We denote this graph as $\text{Repaired}_B(A_m, A_s)$. Then we can establish the following lemma.

Lemma 7. *For every state graphs* A_m *and* A_s, $Repaired_B(A_m, A_s)$ *is bisimulation equivalent to* A_s.

Proof : Here we sketch a brief proof plan. According to the definition of bisimulation equivalence, we only have to check those states that are reachable from the initial states. This means that we do not need to consider states in $A_m - \langle A_m \rangle_{B(A_m,A_s)}$. We can first assume that for some state q_r in $\text{Repaired}_B(A_m, A_s)$ that is reachable from an initial state, there is no q_s in A_s such that $(q_r, q_s) \in B(\text{Repaired}_B(A_m, A_s), A_s)$. There are two cases to analyze. The first is that

there is a transition (q_r, a, q'_r) that $\text{Repaired}_B(A_m, A_s)$ can do at q_r to transit to q'_r but A_s cannot do at any q_s to transit on input a to a state q'_s with $(q'_r, q'_s) \in B(\text{Repaired}_B(A_m, A_s), A_s)$.

- Assume that q_r is in $\langle A_m \rangle_{B(A_m,A_s)}$. According to the definition of $B(A_m, A_s)$, there is a q_s in $\langle A_s \rangle_{B(A_m,A_s)}$ such that $(q_r, q_s) \in B(A_m, A_s)$. There are two more cases to analyze.
  - Assume that q'_r is also in $\langle A_m \rangle_{B(A_m,A_s)}$. According to the definition of $B(A_m, A_s)$, there is a (q_s, a, q'_s) in $\langle A_s \rangle_{B(A_m,A_s)}$ such that $(q_r, q_s) \in B(A_m, A_s)$ and $(q'_r, q'_s) \in B(A_m, A_s)$. This violates our assumptions.
  - Assume that $q'_r = [q'_s]$ is in $[A_s] - \langle A_s \rangle_{B(A_m,A_s)}$. According to the construction of $\text{Repaired}_B(A_m, A_s)$, transition $(q_r, a, [q'_s])$ is there because we have a transition (q_s, a, q'_s) in A_s with $(q_r, q_s) \in B(A_m, A_s)$ and $q'_r = [q'_s]$. This also violates the assumptions.
- The case that q_r is in $[A_s] - \langle A_s \rangle_{B(A_m,A_s)}$ can be proved in a symmetric way.

The "*vice versa*" part is symmetric and is that there is a transition (q_s, a, q'_s) that A_s can do at q_s but $\text{Repaired}_B(A_m, A_s)$ cannot match at any q_r. This case can be proven in a symmetric way. Thus the lemma is proven. ■

The following lemma shows the complexity of the algorithm.

Lemma 8. *$\text{Repaired}_B(A_m, A_s)$ is constructible in PTIME.*

Proof : According to the classical bisimulation checking algorithm [15], $B(A_m, A_s)$, $\langle A_m \rangle_{B(A_m,A_s)}$, and $\langle A_s \rangle_{B(A_m,A_s)}$ can all be calculated in PTIME. It is easy to see that $[A_s] - \langle A_s \rangle_{B(A_m,A_s)}$ can also be computed in PTIME. Finally, to disable $A_m - \langle A_m \rangle_{B(A_m,A_s)}$ and to glue $[A_s] - \langle A_s \rangle_{B(A_m,A_s)}$ to $\langle A_m \rangle_{B(A_m,A_s)}$, there are at most polynomial number of states and transitions to check and to work on. Thus the lemma is proven. ■

Suppose $A_m = (Q_m, P, \mu_m, \Sigma, E_m)$, $A_s = (Q_s, P, \mu_s, \Sigma, E_s)$, $\langle A_m \rangle_{B(A_m,A_s)} = (Q_b, P, \mu_b, \Sigma, E_b)$, and $[A_s] - \langle A_s \rangle_{B(A_m,A_s)} = (Q_d, P, \mu_s, \Sigma, E_d)$. By carefully counting the edit operations, we find that the repair cost suggested by $\text{Repaired}_B(A_m, A_s)$ can be computed as follows.

$$
\begin{aligned}
& |ini(A_m)| - |ini(\langle A_m \rangle_{B(A_m,A_s)})| \\
+ & \left| \left\{ (q_1, a, q_2) \;\middle|\; \begin{array}{r} (q_1 \in ini(A_m) - ini(\langle A_m \rangle_{B(A_m,A_s)}) \wedge (q_1, a, q_2) \in E_m) \\ \vee\, (q_2 \in ini(A_m) - ini(\langle A_m \rangle_{B(A_m,A_s)}) \wedge (q_1, a, q_2) \in E_s) \end{array} \right\} \right| \\
+ & |\{(q_1, a, q_2) \mid q_1 \in Q_b, q_2 \in Q_m - Q_b, (q_1, a, q_2) \in E_m\}| \\
+ & |[A_s] - \langle A_s \rangle_{B(A_m,A_s)}| \\
+ & |\{(q_1, a, [q_2]) \mid \exists (q_1, q') \in B(A_m, A_s)((q', a, q_2) \in E_s \wedge [q_2] \in Q_d)\}| \\
+ & |\{([q_2], a, q_1) \mid \exists (q_1, q') \in B(A_m, A_s)((q_2, a, q') \in E_s \wedge [q_2] \in Q_d)\}|
\end{aligned}
$$

As for the complexity of the aglorithm, it is easy to see that this algorithm only incurs polynomial numbers of set subtractions, graph subtractions, bisimulation computations, and graph edit operations. This justifies that the algorithm is in PTIME and only uses polynomial complexity of memory. Due to page limit, we choose to omit the detailed complexity analysis.

6.4 Suggestions for Repairing Programs

The repairs that we may construct in subsection 6.3 are for model state graphs. The engineers still need to know how such repairs can be used as repair suggestions for their programs. Here we give the following rules for deriving repair suggestions for programs. Again, suppose $A_m = (Q_m, P, \mu_m, \Sigma, E_m)$, $A_s = (Q_s, P, \mu_s, \Sigma, E_s)$, $\langle A_m \rangle_{B(A_m,A_s)} = (Q_b, P, \mu_b, \Sigma, E_b)$, and $[A_s] - \langle A_s \rangle_{B(A_m,A_s)} = (Q_d, P, \mu_s, \Sigma, E_d)$. We also assume that for each state in A_m, we still have the information of its entry statements and exit statements in the original program.

- For every initial state in $A_m - \langle A_m \rangle_{B(A_m,A_s)}$, we suggest to the engineers that for such a state, its entry statements should not be the entry points of the program.
- For each transition from a state q_1 in $\langle A_m \rangle_{B(A_m,A_s)}$ to a state q_2 in $A_m - \langle A_m \rangle_{B(A_m,A_s)}$, we suggest to the engineers that the exit statement for the transition from q_1 to q_2 should be disabled.
- We suggest that a program segment that implements $[A_s] - \langle A_s \rangle_{B(A_m,A_s)}$ should be there.
- For each transition from a state q_1 in $\langle A_m \rangle_{B(A_m,A_s)}$ to a state q_2 in $[A_s] - \langle A_s \rangle_{B(A_m,A_s)}$, we suggest to the engineers that we should change a statement of q_1 to a conditional branch statement that may branch to q_2.
- For each transition from a state q_1 in $[A_s] - \langle A_s \rangle_{B(A_m,A_s)}$ to a state q_2 in $\langle A_m \rangle_{B(A_m,A_s)}$, we suggest to the engineers that we should enter an entry statement of q_2 from an exit statement of q_1.

Such suggestions may not lead to the best repair that the engineers may have in mind. But we feel it is certainly a good mechanical support for some initial ideas in repairing a program.

7 Implementation and Experiment

Our experimental tool ModelRepair ver.0.1 realizes part of our ideas in finding a minimum repair. The tool supports the construction of MCS, the exploration of a repair space in searching for a repair, and repair construction with the PTIME heuristic algorithm. The tool is available at `http://cc.ee.ntu.edu.tw/~val`. To visualize the model, we offer interfaces to convert our graph representations into the GOAL format [17]. The users can thus conveniently see the differences between a repaired model and an original model.

To check how well our algorithm performs, we have also implemented an exploration procedure that searches through the space of edit sequences for a minimum cost repair based on the results in section 5. The search strategy of the procedure is breadth-first. Thus it is guaranteed to find a minimum cost repair if enough time and space are allocated. Also we have designed some strategies to speed up the exploration, including partial order among edit operations. The procedure may still run slowly due to the vast repair space. However it can

Table 1. Performance of ModelRepair ver.0.1

B	A_m			A_s			UB	exploration algorithm					PTIME algorithm				
	$\|A_m\|$	$\|Q\|$	$\|E\|$	$\|A_s\|$	$\|Q\|$	$\|E\|$		$\|A_m\sigma\|$	$\|Q\|$	$\|E\|$	$\|\sigma\|$	time	$\|A_m\sigma\|$	$\|Q\|$	$\|E\|$	$\|\sigma\|$	time
1	11	4	7	14	5	9	5	10	4	6	1	1.79s	14	5	9	5	0.46s
2	9	4	5	15	6	9	10	9	4	5	2	16.6s	9	4	5	4	0.50s
3	4	2	2	7	3	4	3	7	3	4	3	645s	7	3	4	3	0.32s
4	23	9	14	29	11	18	22	$N/A, > 30min$					23	9	14	16	101s
5	18	8	10	20	8	12	18	$N/A, > 30min$					18	8	10	12	11.1s
6	12	4	8	16	5	11	5	Specification inconsistency, 0.53s									
7	12	4	8	15	5	10	5	No repair needed, 0.32s									

B: benchmarks; UB: minimum cost upper-bound predicted with lemma 3;
σ: the corresponding repair; $|Q|$: # states; $|E|$: # transitions; s: seconds;

be used for performance comparison. For interested readers, we have left the procedure to a full version of the paper in our tool website.

We have applied our tool to a few examples. Table 1 summarizes the result of the experiment. All data are collected in Java runtime environment 1.6.0 with Intel Pentium-M 1.6 GHz processor and 512MB RAM. Here 'σ' denotes the repairs we construct. As can be seen, for benchmarks 1 to 5, the PTIME algorithm runs much faster than the repair-space exploration algorithm. For benchmarks 2, 4, and 5, our PTIME algorithm also yields a repair cost lower than the upper-bound predicted by lemma 3 in the column under 'UB.'

For all benchmarks, our heuristic algorithm constructs a repair in less time than the exploration procedure. For benchmarks 1 and 2, our heuristic algorithm constructs repairs with costs greater than the minimum repair costs. But still for benchmarks 2, 4, and 5, the repair costs of our heuristic algorithm are lower than the predicted theoretical upper-bound. In contrast, the exploration procedure did not construct the minimum repairs for benchmarks 3, 4, and 5 in a reasonable amount of time.

8 Conclusion and Future Directions

Our work focuses on the automatic generation of repair suggestions with a cost evaluation that could be useful in controlling the budget for program debugging and preserving the original design intention. We feel that our work could be used as a general foundation for the future research in this direction. One thing is that bisimulation-based repair suggestions may sometimes be based on too strong an assumption. Some program faults may destroy non-trivial bisimulation relations between a model and a specification. In such a case, our algorithm may yield worst-cost repairs. In the future, we may need to investiage what kind of repair suggestions we should make in such a case.

Acknowledgment

We wish to thank Prof. Yih-Kuen Tsay and Mr. Yu-Fang Chen for their helpful suggestions and effort in modifying GOAL to support our implementation and experiments.

References

1. Ball, T., Rajamani, S.K.: Automatically Validating Temporal Safety Properties of Interfaces. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 103–122. Springer, Heidelberg (2001)
2. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers 35(8), 677–691 (1986)
3. Buccafurri, F., Eiter, T., Gottlob, G., Leone, N.: Enhancing Model Checking in Verification by AI Techniques. Artificial Intelligence 112(1), 55–93 (1999)
4. Bunke, H.: On a Relation between Graph Edit Distance and Maximum Common Subgraph. Pattern Recognition Letters 19, 255–259 (1997)
5. Clarke, E., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131. Springer, Heidelberg (1982)
6. Ding, Y., Zhang, Y.: A Logic Approach for LTL System Modification. In: Hacid, M.-S., Murray, N.V., Raś, Z.W., Tsumoto, S. (eds.) ISMIS 2005. LNCS (LNAI), vol. 3488, pp. 435–444. Springer, Heidelberg (2005)
7. Ding, Y., Zhang, Y.: Algorithms for CTL System Modification. In: Khosla, R., Howlett, R.J., Jain, L.C. (eds.) KES 2005. LNCS (LNAI), vol. 3682. Springer, Heidelberg (2005)
8. Fisler, K., Vardi, M.Y.: Bisimulation Minimization in an Automata-Theoretic Verification Framework. In: Gopalakrishnan, G.C., Windley, P. (eds.) FMCAD 1998. LNCS, vol. 1522. Springer, Heidelberg (1998)
9. Griesmayer, A., Bloem, R., Cook, B.: Repair of Boolean Programs with an Application to C. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144. Springer, Heidelberg (2006)
10. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, New York (1979)
11. `http://javaddlib.sourceforge.net/jdd/`
12. Jobstmann, B., Griesmayer, A., Bloem, R.: Program Repair as a Game. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576. Springer, Heidelberg (2005)
13. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
14. Myers, G.J., Sandler, C., Badgett, T., Thomas, T.M.: The Art of Software Testing. Wiley, Chichester (2004)
15. Paige, R., Tarjan, R.E.: Three Partition Refinement Algorithms. SIAM J. 6, 973–989 (1987)
16. `http://www.ifi.unizh.ch/ddis/research/semweb/simpack/`
17. Tsay, Y.-K., Chen, Y.-F., Tsai, M.-H., Wu, K.-N., Chan, W.-C.: GOAL: A Graphical Tool for Manipulating Buchi Automata and Temporal Formulae. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424. Springer, Heidelberg (2007)