Differencing Labeled Transition Systems

Zhenchang Xing[1], Jun Sun[2], Yang Liu[1] and Jin Song Dong[2]

[1] National University of Singapore
{xingzc,liuyang,dongjs}@comp.nus.edu.sg
[2] Singapore University of Technology and Design
sunjun@sutd.edu.sg

Abstract. Concurrent programs often use Labeled Transition Systems (LTSs) as their operational semantic models, which provide the basis for automatic system analysis and verification. System behaviors (generated from the operational semantics) evolve as programs evolve for fixing bugs or implementing new user requirements. Even when a program remains unchanged, its LTS models explored by a model checker or analyzer may be different due to the application of different exploration methods. In this paper, we introduce a novel approach (named SpecDiff) to computing the differences between two LTSs, representing the evolving behaviors of a concurrent program. SpecDiff considers LTSs as Typed Attributed Graphs (TAGs), in which states and transitions are encoded in finite dimensional vector spaces. It then computes a maximum common subgraph of two TAGs, which represents an optimal matching of states and transitions between two evolving LTSs of the concurrent program. SpecDiff has been implemented in our home grown model checker framework PAT. Our evaluation demonstrates that SpecDiff can assist in debugging system faults, understanding the impacts of state reduction techniques, and revealing system change patterns.

1 Introduction

Concurrent programs involve a collection of processes whose behaviors heavily depend on their interactions with other processes and on their reactions to the environment stimuli. The Labeled Transition System (LTS) provides a generic semantic model for capturing the operational semantics of concurrent programs, and is widely used as a basis for automatic software analysis like model based testing [10] or model checking [4]. This semantic model evolves as the program evolves due to bug fixing or implementing new user requirements. A minor syntactic change may lead to significantly different semantic models. For example, a minor change to an atomic step in a concurrent stack program (see Section 3) can lead to very different system behaviors and the violation of critical properties (e.g., linearizability [6]). Even when the program remains syntactically unchanged, its LTS model explored by a model checker or analyzer may be different due to the application of different exploration methods. For example, a model checker may apply partial order reduction [27] or process counter abstraction [21], which can result in a partial LTS compared with the original one.

Identifying the differences in system behaviors of evolving programs is important in debugging and system understanding. Researchers have presented techniques to compute and analyze the changing behavior of programs based on code statements [11],

control flow [2], data flow [14], and symbolic execution [20, 22]. These program representations are not effective in analyzing and verifying the interactions between concurrent processes. Pinpointing differences in the evolving LTSs of a concurrent program can lead to effective analysis of the evolving behaviors of concurrent programs. The underlying assumption is that the evolving LTSs of a program are structurally similar and the structural differences in LTSs can reveal the behavioral changes of a program.

However, computing differences between LTSs is highly nontrivial. The main challenge is how to systematically quantify the similarity of states and transitions and the overall quality of the matching. A state in an LTS can be rather complicated. For concurrent systems, the system configuration has a graph-based structure, in which there are different active processes at different states. The structure of system configuration varies significantly during system transitions. Furthermore, the graph structure of LTSs, such as the incoming and outgoing transitions of states and the transition labels must also be taken into account when comparing two LTSs.

In this paper, we present SpecDiff, an approach to compute and analyze the differences between two evolving LTSs of a concurrent program. The main idea is to represent an LTS as a TAG that encodes the states and transitions in finite dimensional vector spaces, and then exploit the robust graph matching technique to compute an optimal matching of states and transitions between two LTSs. We adopt a modeling language, CSP# [24], for concise behavioral description of concurrent programs. The semantic model of CSP# programs are LTSs. SpecDiff takes as inputs two LTSs of two versions of a CSP# program or of the same program explored with different behavior exploration techniques. It applies GenericDiff framework [28] to compute the differences between two input LTSs. Based on the differences between two LTSs, SpecDiff merges them into a unified model and supports the visualization and query-based analysis of the two LTSs and their differences. Note that our approach is not limited to CSP#, but rather a general method which is capable of identifying the behavioral changes of concurrent programs with LTS-based operational semantic model.

We implement and integrate SpecDiff in our home grown model checker, PAT (Process Analysis Toolkit) [13, 25]. We evaluate the applicability of SpecDiff and its potential benefits in debugging and understanding the evolving behaviors of real-life concurrent programs using three scenarios, in which the LTS changes due to three distinct reasons: 1) the system evolution; 2) the application of partial order reduction; and 3) the application of process counter abstraction. These scenarios demonstrate that SpecDiff can assist in debugging system faults, understanding the impacts of state reduction techniques, and revealing system change patterns.

2 Related Work

One notion commonly used for comparison of system behaviors is refinement. Refinement captures the behavioral relationship between an abstract model of a system (e.g., a specification) and a more detailed model (e.g., an implementation). The correctness of the latter with respect to the former can be established by studying their refinement relationship. Transition-system refinement is commonly defined as trace inclusion or simulation [17]. This definition ensures that if the specification satisfies certain prop-

erty, so does the implementation. This notion of refinement constitutes the foundation of model-based testing [10] and model-based debugging [15].

A symmetric version of the simulation relation is known as bisimulation [16]. Our SpecDiff approach is reminiscent of determining bisimilarity between transition systems. However, bisimilarity requires the behavior of two states to be identical. In contrast, SpecDiff computes a quantitative correspondence value between states, representing how alike they behave. A pair of corresponding states may differ in their system configurations and transitions. Similarly, Girard and Pappas [5] proposed the notion of approximate bisimulation for metric transition systems, whose states and transitions represent quantitative data and computation, such as temperature measurement. The quantitative data constitutes a metric space for measuring the approximate bisimilarity between two metric LTSs. Such approximate bisimilarity allows the possibility of data errors in the analysis of control systems [5]. In our work, the LTSs to be compared do not contain quantitative states and transitions. But SpecDiff encodes the states and transitions of LTSs in finite dimensional vector spaces to quantify their similarities.

If a property is not satisfied, most model checkers will produce a counterexample, which is important in debugging complex systems. For example, Konighofer *et al.* [12] debugs incorrect specifications based on explaining unrealizability using counterexamples. In contrast, SpecDiff analyzes the differences of the evolving LTSs of a concurrent program. It offers more contextual information, since the whole LTSs are compared and differences are highlighted. It can be complementary to counterexample analysis. Furthermore, SpecDiff is also useful in other scenarios, such as assessing the impact of various state space optimization techniques. In such scenarios, model checkers would not provide any counterexamples. However, there is still a need to detect and understand the differences of LTSs.

Program differencing methods [7, 9, 29] have long been used for identifying syntactic and semantic differences between program versions. Person et al. [20] exploit the over-approximating symbolic execution technique to characterize behavioral program differences. Siegel et al. [23] apply model checking and symbolic execution to verify the equivalence of sequential and parallel versions of a program. A recent work by Qi et al. [22] presents a technique to debug evolving programs using symbolic execution and SAT solver. In our work, SpecDiff exploits a robust model differencing framework (i.e., GenericDiff [28]) to compare the evolving LTSs of a concurrent program.

One of the key steps of GenericDiff is to perform a random walk on graph to propagate the correspondence values of node pairs based on graph structure. This process has close connections to the Markov decision process used in [19, 18]. Sokolsky *et al.* [19] compare the LTSs of viruses to classify them into families. Nejati *et al.* [18] compute a similarity measure between Statecharts specifications for finding their correspondences. The goal of our SpecDiff is to detect and analyze the evolving behaviors of a concurrent program, resulted from various reasons.

3 A Motivating Example

We motivate this work with a scenario for an evolving concurrent stack implementation. A concurrent stack is a data structure that provides $push$ and pop operations with

```
1  #define N 2; #define SIZE 2; var H = 0; var HL[N]; Push(i) =
2  τ{HL[i]=H} →
3       ifa(HL[i]==H){ push.i.(H+1){ if(H<SIZE){H++}} → Skip }
4       else{ τ → Push(i) };
5  Pop(i) = τ{HL[i]=H;} →
6       ifa(H==0){ pop.i.0 → Skip }
7       else{
8            τ → ifa(HL[i]≠H){τ → Pop(i)}
9                 else{ pop.i.H{ if(H>0){H−−}} → Skip } };
10 Process(i) = (Push(i) □ Pop(i)); Process(i);
11 Stack()=(|||x:{0..N−1}@Process(x))
```

Listing 1.1. A concurrent stack in CSP# - atomic-**ifa**

the usual LIFO (Last In First Out) semantics for concurrent processes. Herlihy and Wing proposed linearizability [6] as an important correctness criterion for implementations of concurrent data structures. For example, the concurrent stack is linearizable if the projection of the operations in time can be matched to a sequence of operations of a sequential stack. The goal of designing concurrent data structures is to achieve the maximum concurrency yet still preserve the linearizability. The critical design decision is to use suitable locks or synchronization primitives, such as *compare-and-swap* (CAS), *load-linked* (LL) or *store-conditional* (SC) to guarantee the exclusive access of concurrent data structures at the critical points (a.k.a. linearization points). If too many steps are executed atomically, then the throughput of the concurrent data structure is low. If too few steps are executed atomically, then linearizability may be violated.

Trieber [26] proposed a concurrent stack implementation using `CAS` operators. Listing 1.1 shows the algorithm in CSP#. H is the head pointer (being 0 initially) to the top element of stack, $HL[i]$ is a (local) variable of process i to store the value read from the head pointer. The head pointer H is shared by all processes. Each operation tries to update the H until `CAS` operation succeeds. We will further explain the process definitions in Section 4.2. Here it is important to understand that the `CAS` operator is implemented using the variable $HL[i]$, which updates H if the value of H is the same as initially read value in $HL[i]$. The operational semantics of **ifa** (line 4 and 7 of Listing 1.1) is that the condition checking and first event execution of true/false branch are done in one atomic step, which gives the power to simulating the `CAS` operator.

When designing the algorithm, `CAS` operator should be used with care because it requires additional hardware support and reduces the concurrency. In order to maximize concurrency, a modified implementation of the concurrent stack may decrease the atomicity level by changing atomic conditional choice (**ifa**) at line 4 and 7 in Listing 1.1 into a regular conditional choice (**if**). Unfortunately, this change results in the violation of the linearizability of concurrent stack. It is clear that the change of atomicity level of conditional choice affects the correctness of linearizability. However, this minor change results in significantly different system behavior. The LTS of the correct version contains 438 states and 1120 transitions, while the LTS of the faulty version contains 1102 states and 2642 transitions. It is not obvious that why the change of atomicity of conditional choice introduces the fault.

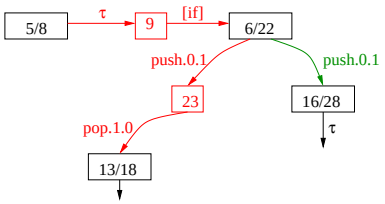

Fig. 1. One incorrect interaction between two processes

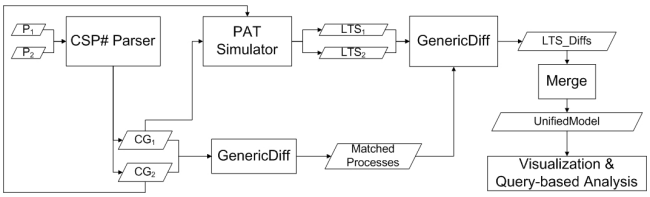

Fig. 2. The architecture of SpecDiff

Our SpecDiff is able to compare the evolving LTSs of the two versions of the concurrent stack. Figure 1 shows one violation of the linearizability of the concurrent stack due to the change of the program. The green states and transitions are reported to be present only in the LTS of the correct version, whereas the red states and transitions are only present in the LTS of the faulty version. SpecDiff reports that the state 6[3] of the correct LTS corresponds to the state 22 of the faulty LTS but the two states behave differently. The state 6 transits to the state 16 by firing a *push.0.1* event in the correct LTS. However, the corresponding state of the state 6, i.e., the state 22 of the faulty LTS does not transit to the corresponding state of the state 16, i.e., the state 28 of the fault LTS by firing a *push.0.1* event. Instead, the state 22 of the faulty LTS transits to the state 23 (a state that is only present in the faulty LTS), by firing a *push.0.1*, from which the system can fire a *pop.1.0* event, which violates the linearizability of the concurrent stack. Essentially, the second process pops nothing after the first process has pushed one item into the stack.

4 The SpecDiff Approach

We begin with an overview of SpecDiff. We then discuss the syntax and semantics of CSP# language. Next, we present how SpecDiff compares the LTSs for detecting the behavioral changes of a concurrent program. Finally, we discuss the visualization and query-based analysis of SpecDiff for inspecting the LTSs and their differences.

4.1 Overview of SpecDiff

Figure 2 presents the architecture of SpecDiff. As a proof of concept, we have implemented SpecDiff in PAT [13] model checker. We adopt CSP# [24] for describing the

[3] The state index is only for illustration purpose.

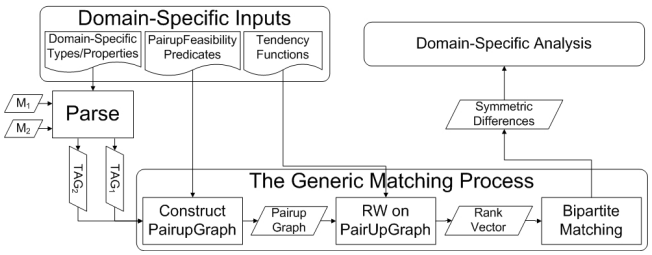

Fig. 3. The architecture of GenericDiff

behavior of concurrent programs, which offers great flexibility in modeling concurrent processes and their interactions.

Given two CSP# programs p_1 and p_2, CSP# parser parses them into two configuration graphs CG_1 and CG_2. The configuration graph is a rooted directed graph, representing the internal syntactic model of a CSP# program. The PAT's simulator that implements the operational semantics of CSP# performs a (bounded) depth-first exploration of configuration graph to generate the LTS model. Given a CSP# program and a particular simulator, the generated LTS is stable across simulations. In our work, the program p_1 and p_2 represent the same program or the two versions of a program. When p_1 and p_2 are the same, different LTSs can be obtained by adopting different simulators that support different behavior exploration methods, such as partial order reduction [27].

The main challenge in comparing the LTSs is quantifying the similarity of states and transitions and the overall quality of the matching. The key idea of SpecDiff is to represent the LTSs as TAGs and exploit the graph matching techniques to determine an optimal correspondence relation over the states and transitions of the input LTSs. More specifically, SpecDiff exploits the GenericDiff framework [28] to compare the evolving LTSs of a CSP# program. First, if the program p_1 and p_2 are different, it applies GenericDiff to compare the configuration graphs (CG_1 and CG_2) of p_1 and p_2 to determine the correspondences between the processes defined in p_1 and p_2, which in turn helps to determine the correspondences between states of two LTSs. Second, it applies GenericDiff to compare the LTSs (LTS_1 and LTS_2) of p_1 and p_2 to determine the correspondences between states and transitions. Based on the matching results of two LTSs, SpecDiff merges the two LTSs into a unified model. It supports visualization and query-based analysis of the two LTSs and their differences.

4.2 Syntax of CSP#

A CSP# [24] program contains the constant and variable definitions, channel definitions and process definitions. Like any program, CSP# programs also evolve. Since the processes are a key factor to determine the state similarity in the corresponding LTSs, given two versions of a program p_1 and p_2, we must first find the syntactic differences between p_1 and p_2. In our work, we compare the configuration graphs CG_1 and CG_2 of p_1 and p_2 to determine the correspondences between processes of p_1 and p_2.

A process is defined as an equation in the following syntax $P(x_1, x_2, \ldots, x_n) = \mathit{ProcessExp}$, where P is the process name, $x_1, x_2, \ldots, x_n$ is an optional list of pro-

cess parameters and $ProcessExp$ is a process expression. A named process may be referenced by its name (with the valuation of the parameters). The process expression defines the computational logic of the process. The following is a BNF description of process expressions [24]. CSP# supports various types of process constructs, including primitives, event prefixing, channel communication, hiding, and various process compositions. CSP# parser parses a CSP# program into a configuration graph $CG(V, E)$, where the vertex set V contains the processes defined in the program and the edge set E contains the composition relations between processes.

$$\begin{aligned} P = \; & Stop \mid Skip \mid e.x\{prog\} \rightarrow P \mid ch!x \rightarrow P \mid ch?x \rightarrow P \mid P \setminus X \\ & \mid P \; ; \; Q \mid P \; \Box \; Q \mid P \sqcap Q \mid [b]P \mid P \parallel Q \mid P \mid\mid\mid Q \mid P \triangle Q \\ & \mid \; if \; b \; \{P\} \; else \; \{Q\} \mid ifa \; b \; \{P\} \; else \; \{Q\} \mid ref(Q) \end{aligned}$$

The concurrent stack program in Figure 1.1 implements Treiber's lock-free concurrent stack [26] in CSP#. It represents the concurrent stack as a singly-linked list with a head pointer to the top element of the stack and uses `CAS` to modify the value of the head pointer atomically. This program defines two constants (Line 1). N is the number of processes and $SIZE$ is the size bound of the stack. To make the state finite, we bound the size of the stack and the number of processes. Line 2 defines a variable H that records the stack head pointer and a variable HL that records the temporary head value of each process. The process definitions $Push(i)$, $Pop(i)$, $Process(i)$, and $Stack()$ specify the exact behaviors of the concurrent stack.

Figure 4 presents the partial configuration graph of this stack program. The $Stack()$ process is defined as the interleaving ($|||$) of N $Process(i)$. The $Process(i)$ is defined as the sequential composition ($;$) of a choice process ($[]$) and itself (self-loop). The choice process ($[]$) is composed of two choices ($Push(i)[]Pop(i)$). The process $Push(i)$ is defined as a event prefixing process $\tau\{HL[i] = H\} \rightarrow \ldots$. τ is the event name and the statement block attached to this event is a sequential program that is executed atomically together with the occurrence of the event. In this example, it update $HL[i]$ to be H. The process $Push(i)$ behaves like the atomic conditional choice process $ifa(HL[i] == H)\{\ldots\}else\{\ldots\}$ after performing $\tau\{HL[i] = H\}$. If the boolean expression $HL[i] == H$ evaluates to true, then $Push(i)$ behaves like the event prefixing process $push.i.(H + 1)\{...\} \rightarrow Skip$ that performs $push.i.(H + 1)$, updates H, and then terminate. If $HL[i] == H$ evaluates to false, $Push(i)$ behaves like the event prefixing process $\tau \rightarrow Push(i)$. Similarly, process $Pop(i)$ defines the behavior of pop operation (details are omitted in Figure 4 for the sake of clarity).

4.3 Operational semantic of CSP#

The operational semantics of CSP# programs is defined in the form of Structural Operational Semantics (SOS) rules [24]. It extends the operational semantics for CSP [3]. These rules translate a CSP# program into an LTS.

An LTS is a 3-tuple $(S, init, \rightarrow)$, which consists of a set of system configurations, i.e., global states, the initial system configuration $init \in S$, and a set of labeled transition relations $\rightarrow$. In CSP#, a state is composed of two components (V, P) where V is a valuation function mapping a variable name (or a channel name) to its value (or a

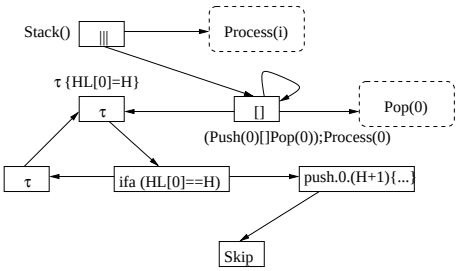

Fig. 4. Stack program configuration graph

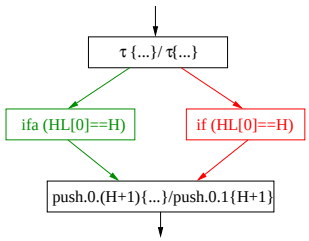

Fig. 5. Partial matching result

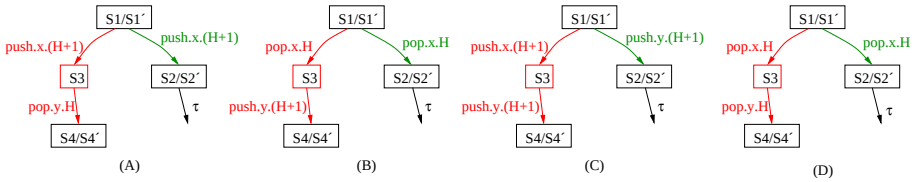

Fig. 6. Four types of incorrect interactions

sequence of items in the buffer) and P is the current process expression. A transition is a labeled directed relation from a source state to a target state. The labeled transition relation $\rightarrow$ conforms to the structural operational semantics of CSP# process constructs. The transition label represents the engaged event. This event has a name and an ordered list (possible empty) of parameter expressions, which captures the information such as process id and the valuation of global variables or channel buffers.

For example, in the LTS of the correct version of the stack program in Figure 1.1, the valuation of global variables H and HL at the state 6 is 0 and $[0, 0]$ respectively. The process expression at the state 6 is $ifa(HL[0] == H)\{push.0.(H + 1)\{\ldots\} \rightarrow Skip\}\ldots;\ Process(0)\ |||\ ifa(H == 0)\{pop.1.0 \rightarrow Skip\}\ldots$. Since $HL[0] == H$ evaluates to true at the state 6, the first process can perform the event prefixing process $push.0.(H + 1)\{\ldots\} \rightarrow Skip$. Consequently, the state 6 transits to the state 16 by a $push.0.1$ transition. The first parameter 0 of the $push.0.1$ event is the id of the first process and the second parameter is the updated head pointer of the shared stack. The valuation of H and HL at the target state 16 is 1 and $[0, 0]$. The process expression at the state 16 is $Skip;\ Process(0)\ |||\ ifa(H == 0)\{pop.1.0 \rightarrow Skip\}\ldots$.

4.4 Comparing configuration graphs and labeled transition systems

SpecDiff applies GenericDiff [28] to compare the evolving configuration graphs and LTSs. GenericDiff is a general framework for model comparison. Given two input models, GenericDiff casts the problem of comparing two models as the problem of recognizing the *Maximum Common Subgraph* of two TAGs. The only step required to apply GenericDiff is to develop the necessary domain-specific inputs (see the architecture of

GenericDiff in Figure 3). The *domain-specific types and properties* specify the TAG that GenericDiff builds when parsing the input model and the characteristic properties of model elements and relations that discriminate their instances. The *pairup feasibility predicates* specify rules that a pair of elements (relations) must satisfy so that they can be paired-up as matching candidates. The *random walk tendency functions* specify the parameters for the random walk process that propagates the correspondence values on graph. GenericDiff reports a *symmetric difference* between two input models, i.e., a set of corresponding model elements and relations in two models and two sets of model elements and relations that are only present in one of the two input models respectively. Due to the space limitation, interested readers are referred to [28] for the technical details about how SpecDiff configures GenericDiff framework.

Figure 5 presents the partially matching results of the configuration graphs of the two versions of the stack program. In this example, the program suffers a minor syntactic change. The atomic conditional choice (green) is only present in the configuration graph of the correct stack program, while the regular conditional choice (red) is only present in the faulty version. All other process constructs are matched.

However, this minor syntactic change to the stack program results in significant semantics changes. Figure 1 presents a violation of linearizability of the concurrent stack among these semantics changes as reported by SpecDiff. Due to the decrease of the atomicity level of conditional choice in the faulty version, the condition checking $if(H == 0)$ will not be executed atomically together with either $pop.i.0 \rightarrow Skip$ (then branch) or $tau \rightarrow ifa(HL[i]! = h)\ldots$ (else branch). Consequently, the second process evaluates $H == 0$ at the state 9 (only present in the faulty LTS), the faulty LTS transits to the state 22. However, at the state 22, before the second process executes $pop.1.0 \rightarrow Skip$, the first process executes $push.0.(H+1)\{\ldots\} \rightarrow Skip$, which update the head pointer of the concurrent stack, i.e., H becomes 1 at the state 23. But the second process is not aware of this update and erroneously execute $pop.1.0 \rightarrow Skip$.

4.5 Analyzing the LTS differences

Given the matching results of two LTSs, SpecDiff merges the two LTSs into a unified model. The unified model is constructed by first creating the matched parts of two LTSs (i.e., corresponding states and transitions) and then applies a sequence of insert operations to create the unmatched states and transitions on the basis of the matched parts of two LTSs. A pair of matched states and transitions appears only once in the unified model. It is important to note that, in our formulation of the LTS similarity, two states (one from each LTS) being matched only indicates that the two states are similar in terms of their characteristic properties and graph structures. As shown in our running example, two matched states (e.g. $6/22$) may still differ in their system configurations (i.e., the valuation of global variables and channels and/or the process expression) and their incoming and outgoing transitions.

To enable an intuitive means of inspecting the differences between the two LTSs, we have developed two types of visualizations for the unified model: normal and fragmented. The normal view shows the unified model in a whole graph. The fragmented view breaks the unified model into a set of disconnected matched and unmatched fragments. A matched (unmatched) fragment is a maximally connected subgraph of matched

(unmatched) states. That is, there are no matched (unmatched) states and transitions in the unified model that could be added to the subgraph and still leave it connected. A unmatched fragment also contains the duplicates of the matched states neighboring with unmatched states. The matched fragments can be hidden in the fragmented view. The detailed state information, i.e., the valuation of global variables and channels as well as the process expression at a state can be inspected in the State Info view or a pop-up window. The visualization supports zooming-in/out and panning the view.

Figure 1 shows partially a normal view of the unified model of the two evolving LTSs of the stack program. The matched states and transitions of two LTSs are shown in black, while the unmatched states and transitions of two LTSs are shown in green and red respectively. In the visualization, the states are indexed with unique ids for illustration purpose. A pair of matched states sid_1 and sid_2 (one from each LTS) is shown in one node labeled sid_1/sid_2. For example, $6/22$ represents that the state 6 of the correct LTS correspondes to the state 22 of the faulty LTS. Note that the state indices have nothing to do with the similarity between states. A pair of matched transitions (one from each LTS) is shown as one edge labeled tl_1/tl_2, tl_1 and tl_2 being the labels of two transitions. When tl_1 and tl_2 are the same, tl_2 is omitted for the sake of clarity. For example, the transition $6/22 \stackrel{pop.1.0}{\rightarrow} 9/17$ represents a pair of matched transitions $6 \stackrel{pop.1.0}{\rightarrow} 9$ and $22 \stackrel{pop.1.0}{\rightarrow} 17$.

In addition to the interactive visual inspection of two LTSs and their differences, SpecDiff stores all the data of two LTSs and their differences in a database. We have defined several queries for detecting behavioral change patterns based on the matching results of two LTSs. For example, one query has been defined to search for pairs of matched states with unmatched same-label transitions. The transitions $6/22 \stackrel{push.0.1}{\rightarrow} 16/28$ and $6/22 \stackrel{push.0.1}{\rightarrow} 23$ shown in Figure 1 is an instance returned by this query.

The visual inspection and query-based analysis complement each other. The visualization provides an intuitive means of inspecting the differences between two LTSs. Query-based analysis scales up to large LTSs. It helps to identify the potentially interesting states and transitions that are worth further investigation. The analysts can then visually explore these states and transitions. In fact, we use the visualization and query-based analysis interleavingly to incrementally build up the knowledge about the two compared LTSs and their differences.

5 Evaluation

In this section, we present our preliminary evaluation of SpecDiff. We focus on the general applicability and the potential benefits of SpecDiff in three scenarios, where the LTS models of concurrent programs change due to three distinct reasons.

5.1 The effectiveness of SpecDiff

We first report our experience in using SpecDiff for debugging and understanding the evolving LTSs of concurrent programs. Three scenarios were illustrated: 1) the evolution of a concurrent stack that results in faulty behaviors; 2) the application of partial

order reduction; 3) the application of process counter abstraction. Note that the programs remain unchanged in the second and third scenarios, but the LTS models actually explored are different due to the application of state space reduction or abstraction techniques.

The evolution of a concurrent stack Concurrent programs are significantly more difficult to design and verify than the sequential ones because process executing concurrently may interleave their steps in many ways, each with a different and potentially unexpected outcome. Our running example demonstrates such a case. A change to the atomicity of conditional choices results in the violation of the linearizability of the concurrent stack. Detecting and analyzing the differences between the correct and faulty LTSs help to debug and understand the evolving behavior of concurrent programs.

In Section 3, we discussed an incorrect interaction between two processes in the faulty concurrent stack (see Figure 1). This incorrect behavior motivated us to define a query searching for pairs of matched states with unmatched same-label transitions. Figure 6 presents four types of incorrect interactions between two processes of the concurrent stack that we have learned from inspecting the SpecDiff results.

Our running example illustrates the first type of incorrect interactions (Figure 6 (A)). The process y pops nothing or an item from the invalid stack top after the process x has pushed one item into the stack. In the second type of incorrect interactions (Figure 6 (B)), one process x executes a pop operation, which updates the head pointer and results in $HL[y] \neq H$, i.e., the temporary head value of the other process y is different from the head pointer. Under this condition, the correct behavior of the process y should perform $\tau \rightarrow Push(y)$ and then update its temporary head value before any $push$ operations. However, due to the non-atomic execution of condition checking $HL[y] == H$ and $push$ operation, the process y pushes one item into the invalid stack top.

In the third type shown in Figure 6 (C), the process x performs a $push$ operation between the condition check $HL[y] == H$ and the $push$ operation of the process y; the process y overrides the item pushed by the process x. In the forth type, the process x performs a pop operation between the condition check $HL[y] == H$ and the pop operation of the process y; the process y pops an item from the invalid stack top.

We also used the PAT model checker to verify the linearizibility of the faulty CSP# concurrent stack program. PAT reports one counterexample $\ldots \rightarrow pop.1.1 \rightarrow pop.0.1$, which represents an instance of the forth type of incorrect interactions between the two processes of concurrent stack. In this particular case, the process 1 pops an item from the stack top (the head pointer H being 1 before the pop operation) such that the stack becomes empty (H being 0). And then the process 0 attempts to pop from the invalid stack top. This counterexample is important in debugging the incorrect program behavior. But it reveals only one case of incorrect interactions. Furthermore, it is not always straightforward to imagine what the corresponding correct behaviors are and what the differences between the correct and incorrect behaviors are. As demonstrated in this case study, our SpecDiff is able to reveal four types of incorrect interactions (see Figure 6) and it is able to offer more contextual information for understanding the evolving behaviors of concurrent programs.

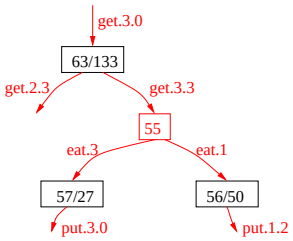

Fig. 7. The impact of partial order reduction

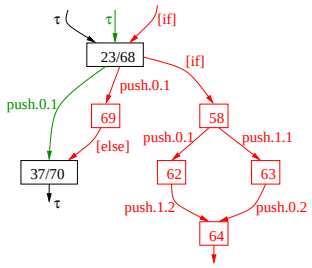

Fig. 8. An example of false positive match

The application of partial order reduction To enable a rigorous correctness proof of a concurrent program, we need to accurately model a concurrent program, for example using formal languages like CSP# [24]. Once the specification of a concurrent program stabilizes, it is often optimized manually or mechanically in order to make verification feasible or efficient [4]. For example, partial order reduction [27] is a technique for reducing the state space to be explored by a model checking algorithm. It exploits the commutativity of concurrently executed independent transitions, which result in the same state when executed in different order. The application of such state reduction techniques can result in the intricate differences in the partial LTS being explored. Identifying these differences helps developers better understand the impact of state space reduction techniques.

In this scenario, we have implemented the classic dining philosophers problem in CSP#, which demonstrates the multi-process synchronization problem in concurrent computing. We simulated two LTSs of this dining philosophers CSP# program with and without partial order reduction respectively. With four philosophers, the LTS obtained without partial order reduction contains 1297 states and 4968 transitions, while the LTS obtained with partial order reduction contains 1214 states and 3396 transitions.

We applied SpecDiff to compare these two LTSs. SpecDiff isolated the 83 states and 1572 transitions that have not been explored when the partial order reduction is in place. Figure 7 presents partially an unmatched LTS fragment. At the state 63 and the state 133, the first philosopher P_1 have grabbed two forks and the third P_3 philosopher have grabbed one fork. Without partial order reduction, there are three ways to proceed, i.e., P_1 eats ($eat.1$ transition, not shown in Figure 7), P_3 grabs another fork ($get.3.3$ transition), or the second philosopher P_2 grabs one fork ($get.2.3$ transition). With partial order reduction, only one way (i.e., P_1 eats) is possible. Consequently, the partial LTS explored with partial order reduction will not consist of the transitions $get.3.3$ and $get.2.3$; the system will not enter the state 55 in which both P_1 and P_3 have grabbed two forks and can eat.

The application of process counter abstraction A parameterized system has finite types of processes, but the number of processes of each type can be very large or even unbounded. Such systems frequently arise in concurrent algorithms and protocols, such

as the classic readers-writers problem and the Java meta-lock algorithm [1]. Process counter abstraction [21] is a common state abstraction technique for analyzing parameterized systems, which groups the processes based on which state of the local finite state machine they reside in. To achieve a finite state abstract system, one can then adopt a *cutoff* number, so that any counter greater than the *cutoff* number is abstracted to w (unbounded number). This yields a sound but incomplete verification procedure - any linear temporal logical property verified in the abstract system holds for all concrete finite-state instances of the system, but not vice versa. In such cases, it is desirable to find plausible change patterns of system behavior as the *cutoff* number changes, since inspecting such change patterns may lead to effective abstraction acceleration and system verification.

Let us start with the classic readers-writer problem. We implemented the readers-writer lock pattern in a parameterized specification in CSP#. The readers-writer lock allows concurrent read access to an object but requires exclusive access for write. It is a synchronization primitive supported by Java version 5 or above and C#. We simulated 20 LTSs of this CSP# readers-writer lock program by setting the *cutoff* number to $i(i = 1..20)$. We applied SpecDiff to compare the consecutive LTSs lts_i and lts_{i+1} and then inspected the differences between lts_i and lts_{i+1} as the *cutoff* number increases.

Let N be the maximum number of readers that can read concurrently. SpecDiff revealed that, as the *cutoff* number i increases by 1, there will be $N - i - 1$ additional fragments in the lts_{i+1}. An additional fragment links two pairs of matched states SP_1 and SP_2 with $i + 3$ unmatched states in between. At SP_1, an unbounded number of readers do not hold the lock and one reader holds the lock, while at the other pair of matched states SP_2, one reader does not hold the lock and an unbounded number of readers hold the lock. From SP_1, the only reader that holds the lock releases the lock (i.e., stopread); the lts_{i+1} transits to an unmatched state where no readers hold the lock. Then, the readers keep acquiring the lock (i.e., startread) until all the readers hold the lock. Finally, one reader releases the lock and the lts_{i+1} reaches to SP_2 where only one reader does not hold the lock. The transitions from SP_2 to SP_1 is in reverse.

We also implemented a parameterized abstract specification in CSP# for Java metalock algorithm [1]. Java metalock plays an essential role in allowing Java to offer concurrent access to objects. Metalocking can be viewed as a two-tiered scheme. At the metalock level, a thread waits until it can enqueue itself on an object's monitor queue in a mutually exclusive manner. We simulated 9 LTSs of this CSP# program by setting the number of threads that can wait at waiting state to $m(m = 2..10)$. We applied SpecDiff to compare the consecutive LTSs lts_m and lts_{m+1}. SpecDiff revealed that, as m increases by 1, lts_{m+1} will have 40 more states and 90 more transitions. There are 10 pairs of matched states, from which lts_{m+1} transits to unmatched states by $getslow$ transitions (i.e., obtaining an object lock by a slow path). From those unmatched states, lts_{m+1} then transits to other unmatched states until it finally transits back to the 10 pairs of matched states by $request$ transitions (i.e., signaling the request for an object).

5.2 The robustness of SpecDiff

The quantitative similarities of states and transitions are heuristic estimates, based on the characteristic properties of states and transitions as well as the graph structure of

LTSs (see Section 4.4). In this section, we evaluate how good the heuristics of SpecDiff are in matching the corresponding states and transitions in two evolving LTSs.

In principle, the precision and recall metrics are used to evaluate the quality of such matching tasks. Given the total number of matched states (M_{actual}) and the number of matched states reported by SpecDiff ($M_{reported}$), precision is the percentage of the correctly reported matches ($M_{actual} \cap M_{reported}$)/$M_{reported}$ and recall is the percentage of matches reported ($M_{actual} \cap M_{reported}$)/$M_{actual}$. In this work, we have manually examined two compared LTSs to establish the oracle (i.e., M_{actual}) for the analysis. Overall, the precision and recall of SpecDiff is fairly good. In the first scenario, the precision and recall of SpecDiff is 95% and 95% respectively. SpecDiff achieves 100% precision and 100% recall in the second and third scenarios. We attribute this to the rich domain-specific properties and graph structure of the input LTSs.

Figure 8 presents an example of false positive (i.e., erroneous) match of states in the first scenario. SpecDiff reports the state 23 of the correct LTS and the state 68 of the faulty LTS as a pair of corresponding states. However, the state 23 should be matched to the state 58 of the faulty LTS, as the state pair (23/58) can better reflect the violation of the linearizibility of the concurrent stack. It will be an instance of the third type of violations (see Figure 6).

However, as the set of active processes at the state 23 and the state 58 is "too" different, the state 23 and the state 58 are not paired-up as matching candidates. Consequently, SpecDiff matches the state 23 to the state 68 which is one transition (if) away from the state 58. Since the matching of the state 23 and the state 68 is less intuitive for understanding the violation of the linearizibility of concurrent stack, we consider it as a false positive match. In the first scenario, such erroneous matches prevent the states from being matched to their "real" counterparts, which consequently results in the false negatives (i.e., missed matches).

6 Threats to Validity

In this work, we ground our discussion on CSP# for modeling the behavior of concurrent programs. We exploit the syntax and structural operational semantics of CSP# to quantify the similarity between the LTSs of a concurrent program. However, the foundational concept of SpecDiff is general, i.e., representing a labeled transition system as a typed attributed graph, quantifying the states and transitions in finite dimensional vector spaces, and exploiting the graph differencing framework to compare the LTSs. Given a modeling language with different syntax and operational semantics, SpecDiff should be applicable as long as the language has LTS-based operational semantics.

The SpecDiff is used to compare the evolving LTSs of two versions of a program or the LTSs of a program explored by different behavior exploration techniques. The underlying assumption is that the structural differences of syntactic models and LTSs of a program can reveal the syntactic and behavioral changes of the program under investigation. However, this assumption does not hold for two arbitrary programs. Two different programs may have the same LTSs. On the other hand, the LTSs being different does not indicate that the two programs must behave differently.

While our preliminary evaluations demonstrate the applicability and potential benefits of SpecDiff, its practical utility still needs further assessments. Scalability is an important challenge to our SpecDiff approach. We are currently exploring a few ways to mitigate the scalability issue. First, we may explore syntactic differences (which could be easy to compute) to guide the comparison of large LTSs. While specification remains unchanged, limiting the depth of search could be one solution. Alternatively, we are considering integrating intuitive visualization technique that allows the user to interactively explore the state space and select which part(s) of the LTSs to differentiate. This would incorporate the human intelligence to guide an interactive differencing process, because the user would have clues about which parts most likely go wrong. Second, we are reviewing the current implementation that compares the LTSs rendered in the GUI. Direct comparison of the internal data structures of LTSs could significantly reduce the execution time and memory consumption. Last but not least, our experiment suggests that often the important differences (e.g. faults) would be reflected in the differences of small sized models. Similar experience has been reported by other verification tools like Alloy [8].

7 Conclusions and Future Work

In this paper, we present SpecDiff for identifying the behavioral changes of concurrent programs with LTS-based semantic model. The main challenge in comparing LTSs lies in how to systematically quantify the similarity of states and transitions of the LTSs and the overall quality of the matching. Our solution is to represent the labeled transitions systems as typed attributed graphs, encodes the states and transitions in finite dimensional vector spaces, and exploits the robust graph matching techniques to determine an optimal correspondence relation over the states and transitions of the input LTSs.

We have developed a proof-of-concept implementation of SpecDiff on the PAT model checker. We evaluated the applicability and the potential benefits of SpecDiff in the evolution and optimization of concurrent programs, written in CSP#, a modeling language for concurrent systems. Our evaluation shows that SpecDiff is able to produce an accurate matching results between the evolving LTSs of a concurrent program. The reported differences are useful in debugging program faults and understanding the behavioral change patterns of concurrent programs.

This work is the first step in exploiting the model differencing techniques to support the development and verification of concurrent programs. Our future work will further develop more types of analysis based on the SpecDiff results. We also plan to extend SpecDiff to compare and analyze real-time systems and web services.

References

1. O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. Ramakrishna, and D. White. An Efficient Meta-Lock for Implementing Ubiquitous Synchronization. In *OOPSLA'99*, pages 207–222, 1999.
2. H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *ISSRE'95*, pages 143–151, 1995.

3. S. D. Brookes, A. W. Roscoe, and D. J. Walker. An Operational Semantics for CSP. Technical report, 1986.
4. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
5. A. Girard and G. Pappas. Approximation metrics for discrete and continuous systems. *IEEE Transactions on Automatic Control*, 52(5):782–798, 2005.
6. M. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Prog. Lang. and Syst. (TOPLAS)*, 12(3):463–492, 1990.
7. S. Horwitz. Identifying the semantic and textual differences between two versions of a program. *SIGPLAN Not.*, 25(6):234–245, 1990.
8. D. Jackson. *Software Abstractions*. MIT Press, 2006.
9. D. Jackson and D. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM'94*, pages 243–252, 1994.
10. J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-Based Software Testing and Analysis with C#*. Cambridge University Press, 2007.
11. J. Jones and M. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE'05*, pages 273–282, 2005.
12. R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications using simple counterstrategies. In *FMCAD'09*, pages 152–159, 2009.
13. Y. Liu, J. Sun, and J. S. Dong. An Analyzer for Extended Compositional Process Algebras. In *ICSE'08 Companion*, pages 919–920, 2008.
14. W. Masri. Fault localization based on information flow coverage. Technical report, AUB-CMPS-07-10, 2007.
15. W. Mayer and M. Stumptner. Model-based debugging – state of the art and future challenges. *Electron. Notes Theor. Comput. Sci.*, 174(4):61–82, 2007.
16. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
17. R. Milner. Operational and algebraic semantics of concurrent processes. pages 1201–1242, 1990.
18. S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and merging of statecharts specifications. In *ICSE'07*, pages 54–64, 2007.
19. S. K. O. Sokolsky and I. Lee. Simulation-based graph similarity. In *TACAS*, pages 426–440, 2006.
20. S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *FSE'08*, pages 226–237, 2008.
21. A. Pnueli, J. Xu, and L. Zuck. Liveness with (0, 1, infty)-Counter Abstraction. In *CAV'02*, pages 107–122, 2002.
22. D. Qi, A. Roychouhury, Z. Liang, and K. Vaswani. Darwin: an approach for debugging evolving programs. In *FSE'09*, pages 33–42, 2009.
23. S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *ISSTA'06*, pages 157–168, 2006.
24. J. Sun, Y. Liu, J. S. Dong, and C. Q. Chen. Integrating Specification and Programs for System Modeling and Verification. In *TASE'09*, pages 127–135, 2009.
25. J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV 2009*, pages 702–708, Grenoble, France, June 2009.
26. R. K. Treiber. Systems Programming: Coping with Parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.
27. A. Valmari. Stubborn Set Methods for Process Algebras. In *PMIV'96*, pages 213–231, 1996.
28. Z. Xing. Genericdiff: A general framework for model comparison. Technical report, National University of Singpore, 2011. http://www.comp.nus.edu.sg/˜pat/publications/gendiff.pdf.
29. W. Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7):739–755, 1991.