

SpecDiff: Debugging Formal Specifications

Zhenchang Xing, Jun Sun, Yang Liu and Jin Song Dong
School of Computing
National University of Singapore
{xingzc, sunj, liuyang, dongjs}@comp.nus.edu.sg

ABSTRACT

This paper presents our SpecDiff tool that exploits the model differencing technique for debugging and understanding evolving behaviors of formal specifications. SpecDiff has been integrated in the Process Analysis Toolkit (PAT), a framework for formal specification, verification and simulation. SpecDiff is able to assist in diagnosing system faults, understanding the impacts of specification optimization techniques, and revealing the system change patterns.

Categories & Subject Descriptors: D.2 SOFTWARE ENGINEERING; D.2.5 Testing and Debugging, Debugging aids
General Terms: Algorithms, Languages, Reliability, Verification
Keywords - Model Differencing, Model Checking, PAT

1. MOTIVATION

Like programs, specifications may evolve for a variety of reasons, such as fixing bugs, supporting new user requirements, or changing application contexts. Modern specification languages are getting more and more sophisticated in order to capture intricate system behaviors. A minor syntax change may lead to significant changes of system behavior and hence different model checking results. When an evolving specification behaves unexpectedly, it is often challenging to figure out what is wrong and why. Furthermore, the behavior of a specification may change, even when the specification remains unmodified. For example, in the setting of model checking, a model checker may apply state reduction and abstraction techniques that affect the behavior of a specification. In such cases, syntax-based analysis, such as [9] offers little help.

It is desirable to develop systematic semantic differencing methods to detect and analyze the evolving behaviors of formal specifications. For example, Labeled Transition System (LTS) is a generic semantic model to capture operational semantics of specification languages. Pinpointing the differences in the LTSs can help developers understand evolving system behaviors precisely and thus lead to effective system analysis. However, existing program analysis and debugging techniques [2,3,4], which are based on control-flow, data-flow or symbolic execution, etc., cannot be directly applied to the comparison and analysis of LTSs.

Semantic differencing of LTSs is to generate likely mappings of states and transitions between two LTSs. This task is highly nontrivial. First, a state in a LTS that captures necessary information of a system configuration can be rather complicated. For concurrent hierarchical systems, a state itself has a graph-based structure, in which there are different active processes at different levels. The structure of state varies significantly during system transitions. Second, 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 mapping states. Third,

a LTS often has a very large or infinite set of states, it is thus important that a semantic differencing technique can offer helpful information even if only part of the LTS is available.

In this paper, we present our SpecDiff tool for systematically detecting and analyzing the differences between the LTSs (i.e., behaviors) of two versions of a specification or of the same specification with different optimization/abstraction techniques.

2. THE SPECDIFF TOOL

Our SpecDiff approach has been detailed in [7]. As a proof of concept, we have integrated the SpecDiff tool in the Process Analysis Toolkit (PAT) [6]. We have evaluated the SpecDiff tool with specifications written in the CSP# (Communicating Sequential Process #) language [5].

CSP# [5] is one of the specification languages supported by PAT. CSP# combines low-level programming constructs (e.g., C# language) with high-level process constructs. A CSP# specification may consist of processes, events, variables, channels, and C# programs. The low-level programs are used to manipulate complex data structures, while the rich set of process constructs are used to define/compose processes and capture complex (concurrent) control flow.

The semantics of CSP# is defined in the form of structural operational semantics, which translates a CSP# specification into a LTS. A LTS is a 3-tuple $(S, init, \rightarrow)$, which consists of a set S of global states, i.e., system configurations, the initial state $init$, and a set \rightarrow of labeled transition relations. A state in CSP# is itself a 3-tuple (V, C, P) where V is the valuation of global variables, C is the content of channels, and P is the current active-process expression (which is presented as a system configuration graph at runtime). We use the Simulator of PAT to generate the LTS of a specification.

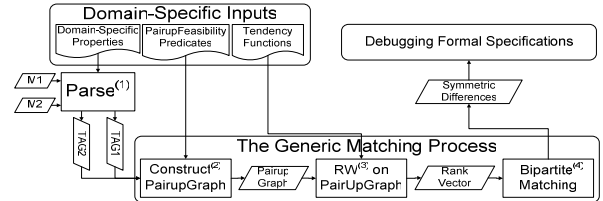


Figure 1 The architecture of GenericDiff framework

Given two LTSs, generated by simulating two versions of a specification or a specification with different optimization and abstraction techniques, SpecDiff configures GenericDiff [8] to detect the differences between the two LTSs. GenericDiff is a framework for model comparison, which can be easily deployed to a wide class of models that can be represented as graphs.

Figure 1 presents the architecture of *GenericDiff*. *GenericDiff* takes as input two models to be compared and the specifications of model properties and syntax in terms of domain-specific properties, pairup feasibility predicates, and random walk

tendency functions. *GenericDiff* parses⁽¹⁾ the input models (in this work LTSs or system configuration graphs of states) into Typed Attributed Graphs (TAGs), according to the metamodel of the models to be compared. It then constructs⁽²⁾ a PairupGraph, i.e., a product of two model graphs, which encodes the graph structure of two models to be compared. Next, *GenericDiff* performs a random walk⁽³⁾ on the PairupGraph, which is an iterative process that propagates the similarity value from node pair to node pair based on graph structure. The random walk on the PairupGraph outputs a rank vector of graph node pairs, each of which is assigned a numerical correspondence measure, i.e., a measurement of the quality of the match it represents. *GenericDiff* builds⁽⁴⁾ a bipartite graph from this rank vector of node pairs and selects an optimal matching⁽⁴⁾ using a stable-marriage algorithm [1].

Given two compared models, SpecDiff constructs a unified model based on the differencing result reported by *GenericDiff*. SpecDiff supports two types of visualizations of the unified model. The normal visualization shows the unified model in a whole graph. The fragmented visualization shows the unified model in a set of disconnected matched and unmatched fragments, i.e., maximally connected subgraphs of matched (unmatched) elements. In both types of visualizations, SpecDiff highlights the unmatched elements and relations in two compared models in green and red respectively. SpecDiff supports the interactive exploration (e.g., zoom-in/out, state info pop-ups, etc.) of two compared models and their differences.

Figure 2 presents our SpecDiff tool. It shows the differences between the LTSs of two versions of a current stack specification (one version uses atomic conditional choice; the other uses regular conditional choice). The main panel shows the differences between the two LTSs in the fragmented visualization. The bottom-left view summarizes these differences in a table. The

bottom-right view depicts the differences between the system configuration graphs of a pair of matched states (e.g., 37/70) in the normal visualization. In this example, inspecting these differences helps to identify the unexpected transitions that lead to the violation of linearizability when regular conditional choice is used for specifying concurrent stack.

REFERENCES

- [1] D. Gale and L.S. Shapley. College admissions and the stability of marriage. *American Math.* 69(1):9-15, 1962.
- [2] J.A. Jones and M.J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. *ASE*, 2005, pp. 273-283.
- [3] W. Masri. Fault localization based on information flow coverage. Technical Report, AUB-CMPS-07-10, 2007.
- [4] D. Qi, A. Roychouhury, Z. Liang and K. Vaswani. Darwin: an approach for debugging evolving programs. *FSE*, 2009, pp. 33-42.
- [5] J. Sun, Y. Liu, J.S. Dong and C.Q. Chen. Integrating specification and programs for system modeling and verification, *TASE*, 2009, pp. 127-135.
- [6] J. Sun, Y. Liu, J.S. Dong and J. Pang. PAT: Towards flexible verification under fairness. *CAV2009*, pp. 702-708.
- [7] Z. Xing, J. Sun, Y. Liu and J.S. Dong. Debugging evolving system behaviors with SpecDiff. Technical Report, NUS, 2009.
- [8] Z. Xing. *GenericDiff: A general framework for model comparison*. Technical Report, NUS, 2009.
- [9] W. Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.* 21(7):739-755, 1991.

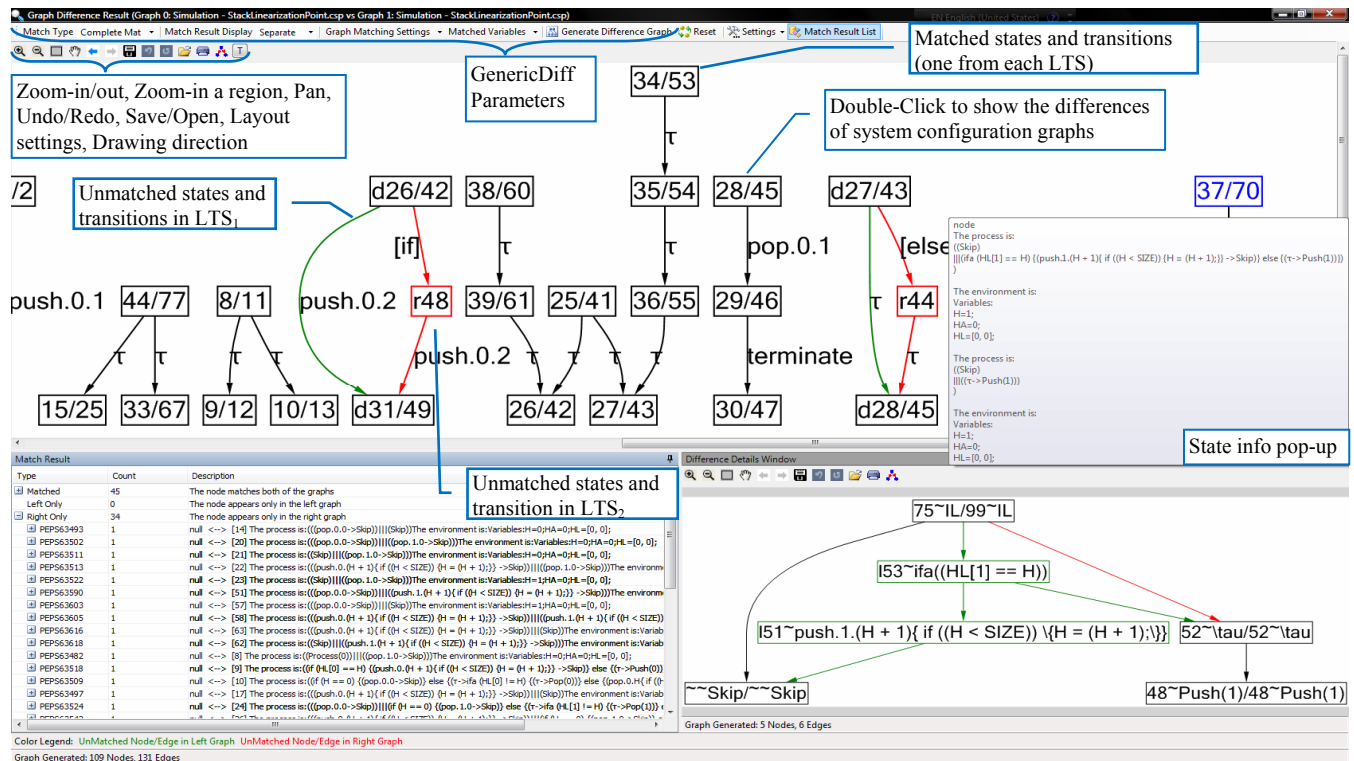


Figure 2 The SpecDiff tool