

# $\chi$ SUDS-SDL: A Tool For Diagnosis and Understanding Software Specifications

J. Jenny Li, Saul London, Plinio Vilela, and J. Robert Horgan  
Telcordia Technologies (formerly Bellcore),  
445 South Str, Morristown NJ 07960-6438  
{jjli | saul | vilela | jrj}@research.telcordia.com

## 1 Introduction

Available statistical data shows that the cost of repairing software faults rises dramatically in later development stages. It is important to have reliable software specifications. In particular, as the technology of direct code generation from specification becomes more popular in practice, reliable software specifications become more important. Automatic code generation allows developers to write specifications directly, without going through coding stage, which promises to reduce cost significantly. However, many issues remain to be resolved before programming on the specification level can become a reality. The goal of this work is to help pave the road so that writing specification is made easier with the aid of software engineering tools. Our toolsuite,  $\chi$ Suds-SDL speeds up the process of understanding, debugging and re-testing software specifications, the three costly tasks in writing software specifications.

Software source-code level programming often utilizes many tools such as debuggers, purifiers, testers, etc. As the programming moves up to the level of software specification, a set of new tools on the specification level will be required. Without proper engineering tools, direct code generation can only be used once with the first version and the debugging and enhancement have to be done on the source code level, because it is difficult to do it on the specification level without tools. It is just like the situation of the beginning of the Fortran language. At that time, people sometimes wrote code in Fortran, and modified the generated assembly code. Our set of software tools aids writing software on the specification level directly. We use Specification and Description Language (SDL) as a specification language.

To illustrate our techniques, we will focus on the  $\chi$  Software Understanding and Diagnosis System ( $\chi$ Suds) for SDL that is under development at Telcordia Technologies (formerly Bellcore) as an aid to understanding and developing software specifications.  $\chi$ Suds uses coverage testing data to provide insights for specification understanding and diagnosis. It first creates a flow graph of the specification, thus laying out its execution structure. It then instruments the program to collect execution trace. The trace file records how many times a given part of the specification, such as a process,

a transition, a decision, a state inputs, or a data flow, has been exercised in each simulation of the specification.  $\chi$ Suds contains a suite of tools:  $\chi$ Slice,  $\chi$ Vue,  $\chi$ Prof, and  $\chi$ ATAC. This paper elaborates the underlying theory and illustrates the usage of these tools.

## 2 $\chi$ Slice and $\chi$ Vue: Debugging and Feature Identification Tools

Debugging tools for software specifications are rare. Checking specification correctness is often done using verification techniques such as model checking, which searches through all reachable states. We believe such an approach is similar to exhaustion testing. Due to market time pressure, sometimes it is impossible to do an exhaustion test, which often requires 100% of the reachable state coverage. Moreover, even if the time allows for verification, none of these techniques automatically locate the bugs as we have done using our tools. Fixing bugs on the specification level is a significant part of the specification development and is often the highest priority activity. Tracking down bugs in unfamiliar specification is a challenge to many specification developer's skills. The  $\chi$ Slice tool is able to help with this task.

$\chi$ Slice uses the concept of an *execution slice* of a specification, which is the part of the specification executed by a particular test case. We assume that the parts covered by successful test cases are bug-free and the bugs are residing in the parts of the specifications that are covered by the failed test cases. Therefore, finding the common parts of the failed test cases and subtracting the parts in the succeeded test cases will help the users to identify bugs.

The original approach of  $\chi$ Slice is to slice the software program. We use this technique to slice software specifications. Each execution slice now is one simulation run of a specification. We refer to each simulation run as an execution slice of the specification. One difference between an actual program and a specification is that specifications allow informal text such as "display the product list". We treat informal text as a regular statement because the tool keeps track of the number of times a statement is reached but not its content. Informal text does not affect the slicing information collection. We were able to treat a specification like a program in our diagnostic analysis.

The only effect of the informal text is that we cannot use the actual execution, but only a simulation. We have developed our own specification simulation tool. We can also use existing commercial tools such as Telelogic SDT and Verilog ObjectGeobe. Each test case in this content is one simulator execution of the specification.

The same technique, execution slicing, can also be used to locate features of the specification. Any non-trivial software system provides many different, but related, *features* to its users. For a telephone switch, features might include call forwarding, call waiting, speed dialing and line diagnostics. For a person new to a project, it can be very difficult to understand where specific features are specified, and thus it is difficult for them to make changes or fix bugs. Furthermore, since the feature is rarely specified in one place, it is difficult to anticipate the effect of changing a specific part of the specification.

We observed that some fractions of the specification are common among features. Other parts are feature-specific and are only executed when the user exercises that specific feature. We find the feature-specific specification components and map them to their corresponding features.

### 3 $\chi$ Prof Performance Tool

It is important to identify performance bottlenecks of software specifications. It can be used to guide the redesign of the system when the specification is a design document.

Designers often want to have optimal performance for the system they design. It seems to be inevitable that any successful system will be stressed by larger and larger data sets until performance bounds are encountered. Execution profiles, showing how much time is spent in each function or subroutine, are the common ways of understanding such performance limitations.

Many profiling tools are available for a wide range of programming languages and environments. There is no such tool for software specifications. The  $\chi$ Prof tool leverages the coverage data already collected in  $\chi$ Suds to provide a simple kind of profiling that counts the number of times each transition is executed instead of counting execution time. While transitions may differ in their execution times, especially for time-consuming system calls, transition counting tends to be a reliable guide to system performance stress points. Execution time profiling is usually done at the process level to keep down the overhead of processing system times. Execution time in SDL states could include waiting time for system user inputs, which can be very large when the user is taking a break. It does not make sense to consider this delay as system performance bottleneck. Transition

counting can point more directly to the exact point in the specification that most impacts performance.

### 4 $\chi$ ATAC Coverage Testing Tool

Once a specification has been written, it needs to be verified. Again, we view verification as testing. In some cases, the specification is not completely new. It evolves from a previous version. It is usually not practical to re-verify the entire specification after a change, which may amount to less than 10% of the specification. Developers need to focus on the part of the specification they have, in fact, changed and then develop or re-use tests that cover these parts as quickly as possible.

The  $\chi$ ATAC component of  $\chi$ Suds helps with this verification process by prioritizing specification transitions, states or decisions. Some transitions or states dominate others in the same function in that, if the dominating one is covered by a test, then many others must also be covered. The dominating ones are found by analyzing the control graph of the specification process. Thus the dominating ones are good places for the users to start in writing verification test cases; if they are covered then coverage of the specification can be greatly increased with just a few test cases[1].

### 5 Concluding Observations

Automatic fault localization in software specifications is a recent technology. The concept of slicing a program statically to find an executable subset was originated by Weiser [2]. Dynamic slicing, based on actual execution of the program, and the idea of using differences between slices from different test cases was described by Collofello and Cousin [3]. We expanded these technologies to the software specification level for writing more reliable specifications efficiently. As we have mentioned,  $\chi$ Vue and  $\chi$ Slice implement this idea. Debugging and performance profiling is rarely mentioned in the field of software specification even though early fault detection can cut costs.  $\chi$ ATAC and  $\chi$ Prof provide the techniques and tools. We hope that the tool suite will make a significant contribution to improve reliability and efficiency of software specification development, especially programming on the specification level in the telecommunications industry [4].

### References

- [1] Agrawal, H., "Dominators, Super Blocks, and Program Coverage" *Conference Record of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, Portland, Oregon, January 1994, pp. 25-34
- [2] Weiser, M., "Programmers Use Slices When Debugging", *Communications of the ACM*, Vol. 25, No. 7, July 1982, pp. 446 - 452.
- [3] Collofello, J. and Cousin, L., "Towards automatic software fault location through decision-to-decision path analysis", *Proceedings National Computer Conference*, 1987, pp. 539 - 544
- [4] [www.xsuds.com](http://www.xsuds.com)