N93-17521

DOMAIN-SPECIFIC FUNCTIONAL SOFTWARE TESTING: A PROGRESS REPORT

522-61
136896

P. 4

Uwe Nonnenmann
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
un@research.att.com

1 Introduction

Software Engineering is a knowledge intensive activity that involves defining, designing, developing, and maintaining software systems. In order to build effective systems to support Software Engeneering activities, Artificial Intelligence techniques are needed. The application of Artificial Intelligence technology to Software Engineering is called Knowledge-based Software Engineering (KBSE) [Lowry & Duran, 1989]. The goal of KBSE is to change the software life cycle such that software maintenance and evolution occur by modifying the specifications and then rederiving the implementation rather than by directly modifying the implementation. The use of domain knowledge in developing KBSE systems is crucial.

Our work is mainly related to one area of KBSE that is called automatic specification acquisition. One example is the WATSON prototype [Kelly & Nonnenmann, 1991] on which our current work is based. WATSON is an automatic programming system for formalizing specifications for telephone switching software mainly restricted to POTS, *i.e.*, *plain old telephone service*.

Other examples of such systems are IDeA and Ozym. The Intelligent Design Aid (IDeA) [Lubars & Harandi, 1987] performs knowledge-based refinement of specifications and design. IDeA gives incremental feedback on completeness and consistency using domain-specific abstract design schemas. The idea behind Ozym [Iscoe *et al.*, 1989] is to specify and implement applications programs for non-programmers and non-domain-experts by modeling domain knowledge.

However, despite two decades of moderately successful research, there have been few practical demonstrations of the utility of Artificial Intelligence techniques to support Software Engineering activities [Barstow, 1987] other than such prototypes as mentioned above. Our current approach differentiates itself from these other approaches in two antagonistic ways: On the one hand, we address a large and complex real-world problem instead of a "toy domain" as in many research prototypes. On the other hand, to allow such scaling, we had to relax the ambitious goal of complete *automatic programming*, to the easier task of *automatic testing*.

2 KITSS Overview

In the *Knowledge-Based Interactive Test Script System (KITSS)*, we have taken this philosophy and applied it to the task of functional software testing. In functional testing, the internal design and structure of the program are ignored. It corresponds directly to uncovering discrepancies in the program's behavior as viewed from the outside world. This type of testing has been called *black box* testing because, like a black box in hardware, one is only interested in how the input relates to the output. The resulting tests are then executed in a simulated customer environment which corresponds to verifying that the system fulfills its intended purpose.

Tests are by definition correct but not exhaustive. KITSS checks and augments given tests and generates related new ones but does not generate the full specification as in WATSON. KITSS can be seen as performing testing from examples. KITSS' strength lies in its very *domain-specific* approach [Barstow, 1985] and customized reasoning procedures. It will change the software life cycle by modifying the functional tests and then rederiving the system tests which corresponds to finding and eliminating software problems early in the development process as in the KBSE paradigm. Therefore, we designed KITSS to be well integrated into our existing design and development process [Nonnenmann & Eddy, 1991].

KITSS achieves this integration by using the same expressive and unobtrusive input medium, namely *test cases*. They describe in English the high-level details of the external design and are written before coding begins. KITSS also produces the same output as before, executable *test scripts* written in an in-house test automation language. These are low-level descriptions derived from test cases for specific test equipment.

To support this integration, KITSS has a *natural language processor* that is trained in the domain's technical dialect [Jones & Eisner, 1992] and converts the

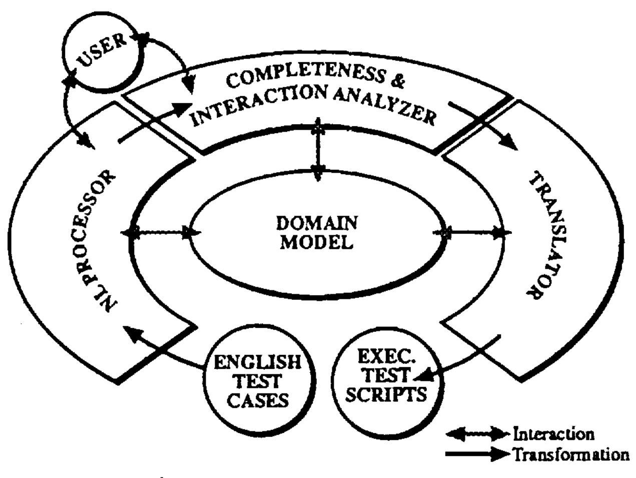

Figure 1: KITSS Architecture

test cases into a formal representation that is audited for coverage and sanity. To accomplish this, KITSS uses a customized theorem prover-based *analyzer* (based on WATSON technology) and a hybrid knowledge base as the *domain model* using both a static terminological logic and a dynamic temporal logic. These two modules have been feasible only due to the domain-specific knowledge-based approach taken in KITSS. Finally, a *translator* takes the corrected test case and converts it from temporal logic into a test script language that can exercise the switch using dedicated test equipment. Figure 1 shows the overall architecture of KITSS.

In summary, KITSS helps the test process by generating more tests of better quality and by allowing more frequent regression testing through automation. Furthermore, tests are generated earlier, *i.e.*, *during* the development phase not *after*, which should detect problems earlier, thus resulting in reduced maintenance costs (for more details on KITSS see [Nonnenmann & Eddy, 1992]).

3 Knowledge Representation Issues

As we used a highly domain-specific approach, the domain model is one of the center pieces of KITSS. In the following section we will highlight the key design decisions made and the knowledge representations chosen.

Testing is a very knowledge intensive task. It involves experience with the switch hardware and testing equipment as well as an understanding of the switch software with its several hundred features and many more interactions. There are many binders of feature descriptions for PBX software, but no concise formalizations of the domain were available before KITSS. The focus of KITSS and the domain model is on an end-user's point of view, *i.e.*, on (physical and software) objects that the user can manipulate. Figure 2 gives an overview of KITSS' domain model.

The *static model* represents all telephony objects, data, and conditions that do not have a temporal extent but may have states or histories. It describes major hardware components, processes, logical resources, the current test setup, the dial plan and the current feature assignments. All static parts of the domain model are implemented in CLASSIC [Brachman *et al.*, 1990], which belongs to the class of terminological logics (*e.g.* KL-ONE).

The *dynamic model* defines the dynamic aspects of the switch behavior. These are constraints that have to be fulfilled during testing as well as the predicates they are defined upon. Objects include predicates, stimuli which can be either primitive or abstract, and observables. Additionally, the dynamic model includes invariants and rules as integrity constraints. Invariants are assertions which describe only a single state, but are true in all states. These are among the most important pieces of domain knowledge as they describe basic telephony behavior as well as the *look & feel* of the switch. Rules describe low-level behavior in telephony. This is mostly signaling behavior.

Representing the dynamic model we required expressive power beyond CLASSIC or terminological logics, which are not well-suited for representing plan-like knowledge. We therefore used the WATSON Theorem Prover, a linear-time first-order resolution theorem prover with a weak temporal logic. This non-standard logic has five modal operators *holds, occurs, issues, begins,* and *ends.* As an action *occurs*, the response to that action may endure until some other action occurs or it may be transient. An enduring actions *begins* and *holds* until, in response to some other action, it *ends.* In the transient case, the switch merely *issues* the response. These modals are sufficient to represent all temporal aspects of our domain. The theorem proving is only tractable due to the tight integration between knowledge representation and reasoning.

In adding the dynamic model, we were able to increase the expressive power of our domain model and to increase the reasoning capabilities as well. The integration of the hybrid pieces did produce some problems, for example, deciding which components belonged in which piece. However, this decision was facilitated because of our design choice to represent all dynamic aspects of the system in our temporal logic and to keep everything else in CLASSIC.

The domain model consists of over 600 domain concepts, over 1,700 domain individuals, and more than 160 temporal axioms.

The domain model was built in the initial phase of the project as the reasoning modules depended on the underlying representations being created first. In this phase the domain model changed constantly as we still enhanced our understanding of the domain. Then, we left the domain model mainly unchanged through the development phases until a milestone was reached. We

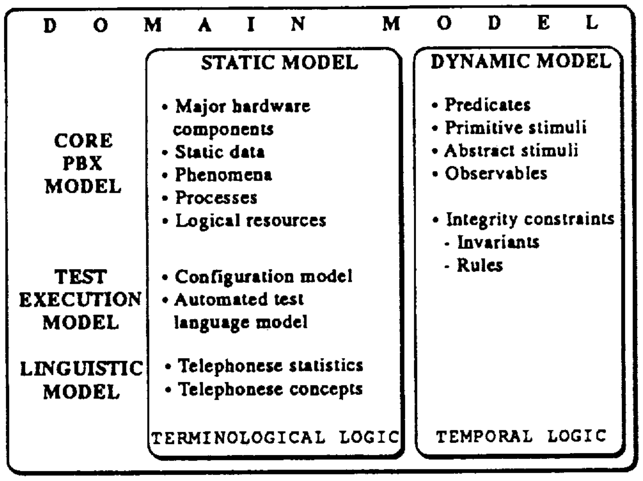

Figure 2: KITSS Domain Model

then typically performed a major revision of the domain model based on problems encountered. The domain model went through three iterations like that and has been stable since. Of course, we continually add new knowledge but the representations are mainly unchanged.

Although we anticipate that the domain model will grow only linear with the number of features covered, we already had great difficulty in acquiring new knowledge and maintaining the existing domain model as both tasks have been done completely manual so far. Therefore, knowledge acquisition and maintenance support is crucial. At least we have gained an understanding on how to design such automated tools. The above experience is the main motivation for another approach included as a proposal in this workshop's proceedings [Hall, 1992].

4 Status

At last year's ASD workshop, we initially reported on KITSS. Since then, we had a major evaluation of the KITSS prototype with the following results.

System execution speed has not been a bottleneck due to continued specialization of the inference capability. However, it is not clear how long such optimizations can avoid potential intractability of the theorem prover. Another result we found was that it is easier to train the natural language processor than it is to achieve coherence in the reasoning audits. The initial scaling phase from the proof-of-principle demo (3 test cases) to the first prototype (38 test cases) was successful but took too long and KITSS was too brittle in general. Although KITSS has been designed from the beginning as an interactive system (the "I" in KITSS), we pushed the *machine initiative* in this scaling phase as far as possible as an experiment. However, this was the limiting factor for rapid progress.

The current schedule is to expand KITSS to cover a few hundred test cases in the next couple of months. To achieve such scaling, we changed some of KITSS' design to make it much more robust. Despite the fact that the natural language processor performed well, we augmented it with a paraphrasing mechanism, *i.e.*, in case the English input cannot be understood, the user can rephrase this input using a paraphrase language based on our temporal logic. When the analyzer encounters problems, it intensely questions the user to explain unclear passages of test cases. This approach has only been possible because KITSS is very responsive. Additionally, we changed all reasoning modules to produce "soft-failures", *i.e.*, in case the system fails to fully understand the test case it still continues to translate user inputs as an *assistant*.

In general, our strategy has shifted from a nearly fully automatic system to one that is much more interactive and might resort to the assistant paradigm occasionally. However, we still have the full KITSS approach in place so that we can improve *e.g.* the analyzer incrementally without any change for the user other than reduced interactions.

5 From Research Into Practice

In 1987, the initial prototype of WATSON was completed. Today, five years later, KITSS is a prototype with a more restricted scope in the same domain although with broader coverage. So where is the big progress? Or in other words: How do we get from a research prototype to a deployed real-world system? Is this "just" technology transfer? Another question might also be: Can we build a research prototype in a toy domain and expect it to work on a real-world problem?

The answers to these questions[1] are not easy, but we would like to give at least partial answers based on our experience.

In scaling from WATSON's toy domain to KITSS' real-world telephony domain we had to address a number of new research issues (which we can just list here without further explanation). For example, we had to extend our temporal logic, restructure the domain model into a hybrid one (static/dynamic), create telephony "micromodels" and understand their interactions, reason with multiple agents instead of single ones, significantly enhance the planning component, understand the "purpose" of inputs to be able to generalize and specialize them, perform more complex nonmonotonic reasoning etc. Additionally, we had to incorporate and customize a natural language module to cover input test cases in English instead of a limited scenario language.

KITSS being so different in all these respects, WATSON cannot be seen as a core system to which we just

[1]These questions are based on a personal discussion with Ron Brachman.

added domain knowledge. For KITSS, we had to basically rewrite and enhance WATSON and add completely new modules. We see KITSS based on technology that WATSON has proven feasible. Therefore, we do not see KITSS as technology transfer at all but as a research project that covers a real-world domain.

Yet despite such research progress, we were still required to further change KITSS' design to achieve practical solutions (see Section 4). No matter how impressive a research prototype looks, we believe there is still a lot of research to be done in order to scale to real-world use.

So: "Can we build a research prototype in a toy domain and expect it to work on a real-world problem?" The answer is: "Probably not".

6 Conclusions

KITSS is a domain-specific system to generate executable functional tests using the KBSE paradigm. Its main difference to previous approaches is that it covers a real-world domain instead of a toy domain. However, therefore the initial goal of automatic programming had to be limited to the easier problem of automatic testing.

KITSS converts input tests into a formal representation interactively with the user. To achieve this, we needed to augment the static domain model represented in a terminological logic with a dynamic model written in a temporal logic. Knowledge acquisition has been performed manually and is a major problem that has not been addressed yet.

In general, scaling from a research prototype to a real-world system involves much additional research before the actual technology transfer can begin. To achieve such scaling, we had to further move toward more user interaction. Although scaling-up remains a hard task, KITSS demonstrates that our KBSE approach chosen for this complex application is feasible.

Acknowledgments

Many thanks go to John Eddy, Van Kelly, Mark Jones, and Bob Hall who also contributed major parts of the KITSS system. Additionally, we would like to thank Ron Brachman for his support throughout the project.

References

Barstow, D.R.: Domain-specific automatic programming. *IEEE Transactions on Software Engineering*, November 1985.

Barstow, D.R.: Artificial Intelligence and Software Engineering. In *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, 1987.

Brachman, R.J., McGuinness, D.L., Patel-Schneider, P.F., Alperin Resnick, L., and Borgida, A.: Living with CLASSIC: When and how to use a KL-ONE-like language. In *Formal Aspects of Semantic Networks*, J. Sowa, ed., Morgan Kaufmann, 1990.

Hall, R.J.: Interactive specification acquisition via senarios: A proposal. In *Proceedings of the AAAI'92 Workshop on Automating Software Design*, San Jose, CA, 1992.

Iscoe, N., Browne, J.C., and Werth, J.: An object-oriented approach to program specification and generation. *Technical Report, Dept. of Computer Science, University of Texas at Austin*, 1989.

Jones, M.A., and Eisner, J.: A probabilistic parser applied to software testing documents. In *Proceedings of the 10th National Conference on Artificial Intelligence*, San Jose, CA, 1992.

Kelly, V.E., and Nonnenmann, U.: Reducing the complexity of formal specification acquisition. In *Automating Software Design*, M. Lowry and R. McCartney, eds., MIT Press, 1991.

Lowry, M., Duran, R.: Knowledge-based Software Engeneering. In *Handbook of Artificial Intelligence, Vol. IV, Chapter XX*, Addison Wesley, 1989.

Lubars, M.D., and Harandi, M.T.: Knowledge-based software design using design schemas. In *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, 1987.

Nonnenmann, U., and Eddy J.K.: KITSS - Toward software design and testing integration. In *Automating Software Design: Interactive Design - Workshop Notes from the 9th AAAI*, L. Johnson, ed., USC/ISI Technical Report RS-91-287, 1991.

Nonnenmann, U., and Eddy, J.K.: KITSS - A functional software testing system using a hybrid domain model. In *Proceedings of the 8th Conference on Artificial Intelligence for Applications*, Monterey, CA, 1992.