PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

Knowledge-Based Functional Testing For Large Software Systems

Uwe Nonnenmann and John K. Eddy
AT&T Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.

Abstract

Automated testing of large embedded systems is perhaps one of the most expensive and time-consuming parts of the software life cycle. It requires very complex and heterogeneous knowledge and reasoning capabilities. The *Knowledge-based Interactive Test Script System* (KITSS) automates functional testing in the domain of telephone switching software. KITSS uses some novel approaches to achieving several desirable goals. Telephone feature tests are specified in English. To support this KITSS has a statistical parser that is trained in the domain's technical dialect. KITSS converts these tests into a formal representation that is audited for coverage and sanity. To accomplish this, KITSS uses a customized theorem prover-based inference mechanism and a hybrid knowledge base as the domain model that uses both a static terminological logic and a dynamic temporal logic. Finally, the corrected test is translated into an in-house automated test language that exercises the switch and its embedded software. This paper describes and motivates the approach taken and also provides an overview of the KITSS system.

1 Functional Testing Problem

There is an increasing amount of difficulty, effort, and cost that is needed to test large software development projects. It is generally accepted that the development of large-scale software with zero defects is not possible. A corollary to this is that accurate testing that uncovers all defects is also not possible [Myers, 1979]. This is because of the many inherent problems in the development of large projects [Brooks, 1987]. As just a few examples, a large project provides support for many interacting features, which makes requirements and specifications complex. Also, many people are involved in the project, which makes it difficult to ensure that each person has a common understanding of the meaning and functioning of features. Finally, the project takes a long time to complete, which makes it even harder to maintain a common understanding because the features change through time as people interact and come to undocumented agreements about the real meaning of features.

The consequence of these problems is that programs that do not function as expected are produced and therefore extensive and costly testing is required. Once software is developed, even more testing is needed to maintain it as a product. The major cost of maintenance is in re-testing and re-deployment and not the coding effort. Estimates, as in [Myers, 1976] and [McCartney, 1991], are that at least 50%, and up to as much as 80%, of the cost in the life cycle of a system is spent on maintenance.

We believe that the only practical way to drastically reduce the maintenance cost is to find and eliminate software problems early and within the development process. Therefore, we designed an automated testing system that is well integrated into the current development process [Nonnenmann & Eddy, 1991]. The focus of our system is on "functional testing" [Howden, 1985]. It corresponds directly to uncovering discrepancies in the program's behavior as viewed from the outside world. In functional testing the internal design and structure of the program are ignored. This type of testing has been called *black box* testing because, like a black box in hardware, one is only interested in the input and how it relates to the output. The resulting tests are then executed in a simulated customer environment. This corresponds to verifying that the system fulfills its intended purpose.

KITSS achieves a good integration into the current development process by using the same expressive and unobtrusive input medium (English functional tests) as is used currently as well as generating tests in the existing automated test language as output. Additionally, KITSS checks the tests for consistency with its built-in extensive knowledge base of "telephony".

Therefore, 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 lead to detecting problems earlier. The result is higher quality software at a lower cost.

In this section, we motivated the need for the approach chosen in KITSS. In the next section, we will describe KITSS in more detail.

2 KITSS Overview

The *Knowledge-based Interactive Test Script System* (KITSS) was developed at AT&T Bell Laboratories to reduce the increasing difficulty and cost involved in testing the software of DEFINITY®PBX switches[1]. Although our system is highly domain dependent in its knowledge base and inference mechanisms, the approach taken is a general one and should be applicable to any functional software testing task.

DEFINITY supports hundreds of complex features such as call forwarding, messaging services, and call routing. Additionally, it supports telephone lines, telephone trunks, a variety of telephone sets, and even data lines. At AT&T Bell Laboratories, PBX projects have many frequent and overlapping releases over their multi-year life cycle. It is not uncommon for these projects to have millions of lines of code.

2.1 Testing Process

Before KITSS, the design methodology involved writing **test cases** in English. They describe the details of the external design and are written before coding begins. The cases, which are written by developers based on the requirements, constitute the only formal description of the external functioning of a switch feature. The idea is to describe how a feature works without having coding in mind.

Figure 1 shows a typical test case. Test cases are structured in part by a *goal/action/verify* format. The goal statement is a very high-level description of the purpose of the test. It is followed by alternating action/verify statements. An action describes stimuli that the tester has to execute. Each stimulus triggers a switch response that the tester has to verify (*e.g.*, a specific phone rings, a lamp is lit, a display shows a message etc).

Overall, there are tens of thousands of test cases for DEFINITY. All these test cases are written manually, just using an editor, and are executed manually in a test lab. This is an error prone and slow process that limits test coverage and makes regression test intervals too long.

Some 5% of the above test cases have been converted into **test scripts** written in an in-house test automation language. Tests written in this language are run directly against the switch software. As this software is embedded in the switching system, testing requires large investments in test equipment (computer simulations are not acceptable as they do not address the real-time aspects of the system). Running and re-running test scripts becomes very time consuming and actually controls the rate at which projects are completed.

GOAL:	Activate CF[2] using CF Access Code.
ACTION:	Set station B without redirect notification[3]. Station B goes offhook and dials CF Access Code.
VERIFY:	Station B receives the second dial tone.
ACTION:	Station B dials station C.
VERIFY:	Station B receives confirmation tone. The status lamp associated with the CF button at B is lit.
ACTION:	Station B goes onhook. Place a call from station A to B.
VERIFY:	No ring-ping (redirect notification) is applied to station B. The call is forwarded to station C.
ACTION:	Station C answers the call.
VERIFY:	Stations A and C are connected.

Figure 1: Example of a Test Case

Although an improvement over the manual testing process, test automation has several problems. The current tools do not support any automatic semantic checking. The conversion from test case to test script takes a long time and requires the best domain experts. There are only limited error diagnosis facilities available as well as no automatic update for regression testing. Also, test scripts are cluttered with test language initialization statements and are specific to switch configurations and software releases. Test scripts lack the generality of test cases, which are a template for many test scripts. Therefore, test cases are easier to read and maintain.

2.2 KITSS Architecture

KITSS takes English test cases as its input. It translates all test cases into formal, complete functional test scripts which are run against the DEFINITY switch software. To make KITSS a practical system required novel approaches in two very difficult and different areas.

First, a very informal and expressive language needed

[1]A PBX, or private branch exchange, switch is a real-time system with embedded software that allows many telephone sets to share a few telephone lines in a private company.

[2]CF is an acronym for the call-forwarding feature, which allows the user to send his/her incoming calls to another designated station. The user can activate or deactivate this feature by pressing a button or by dialing an access code.

[3]Redirect notification is a feature to notify the user about an incoming call when he/she has CF activated. Instead of the phone ringing it issues a short "ring-ping" tone.

to be transformed into formal logic. Test cases are written in English. While English is undeniably quite expressive and unobtrusive as a representation medium, it is difficult to process into formal descriptions. It also requires theoretically unbounded amounts of knowledge to satisfactorily resolve incompleteness, vagueness, ambiguity, etc. In practice, however, test cases are written in a style that is considerably more restrictive than most English text. The test case descriptions are circumscribed in terms of the vocabulary and concepts to which they refer. Syntactic and semantic variations do occur, but the language is a technical dialect of English, a naturally occurring *telephonese* language that is less variable and less complex. These limits to a specific domain and style make it possible to transform the informal *telephonese* representation into a formal one.

Second, incomplete test cases needed to be extended. Even though humans find it easier to write test cases in natural language as opposed to formal language, they still have difficulties specifying tests that are both complete and consistent. They also have difficulties identifying all of the interactions that can occur in a complex system. This is analogous to the difference between trying to define a word and giving examples of its use. Creating a good definition, like creating a complete test case with all the details, is usually the more challenging task; giving word-usage examples, like describing a test case in general terms, is easier. Therefore, the input test cases need to be translated into a formal representation and then analyzed to be corrected and/or extended.

Both tasks have been attempted for more than a decade [Balzer *et al.*, 1977] with only limited success. Most difficulties arise because of the many possible types of imprecision in unrestricted natural language specifications, as well as by the lack of a suitable corpus of formalized background knowledge to guide automated reasoning tools for most application domains.

To address these two difficulties (see also [Yonezaki, 1989]), KITSS provides a natural language processor that is trained on examples of the *telephonese* sub-language using a statistical approach. It also provides a completeness and interaction analyzer that audits test coverage. However, these two modules have been feasible only due to the *domain-specific* knowledge-based approach taken in KITSS [Barstow, 1985]. Therefore, both modules are supported by a hybrid knowledge-base (the "K" in KITSS) that contains a model of the DEFINITY PBX domain. Concepts that are used in telephony and testing are available to both processes to reduce the complexity of their interpretive tasks. If, for example, a process gets stuck and cannot disambiguate the possible interpretations of a phrase, it interacts (the "I" in KITSS) with the test author. It presents the context in which the ambiguity occurs and presents its best guesses and asks the author to pick the correct choice. Finally,

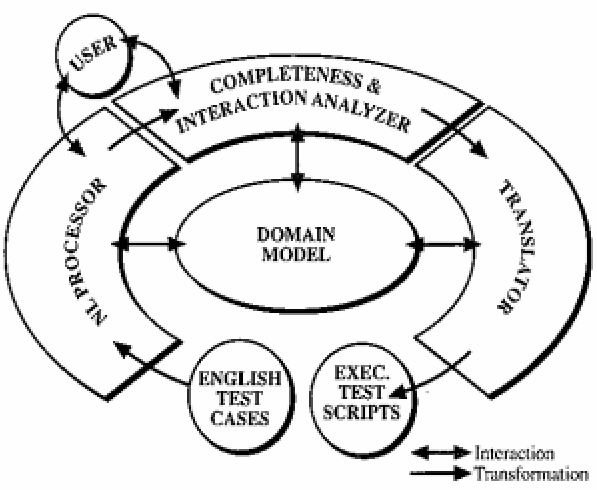

Figure 2: KITSS Architecture

KITSS also provides a translator that generates the actual test scripts (the "TS" in KITSS) from the formal representation derived by the analyzer.

The two needs described above led to the architecture shown in Figure 2. It shows that KITSS consists of four main modules: the domain model, the natural language processor, the completeness and interaction analyzer, and the translator. The domain model (see Section 3) is in the center of the system and supports all three reasoning modules (see Section 4).

3 Domain Knowledge

A domain model serves as the knowledge base for an application system. 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 binders full of papers that describe the features of DEFINITY PBX software, but no concise formalizations of the domain were available before KITSS. One of the core pieces of KITSS is its extensive domain model. 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.

The KITSS domain model consists of three major functional pieces (see Figure 3):

Core PBX model: It is split into two major parts. The static model is used by all reasoning modules. The dynamic model is used mainly by the analyzer.

Test execution model: It includes details about the current switch configuration and all the necessary

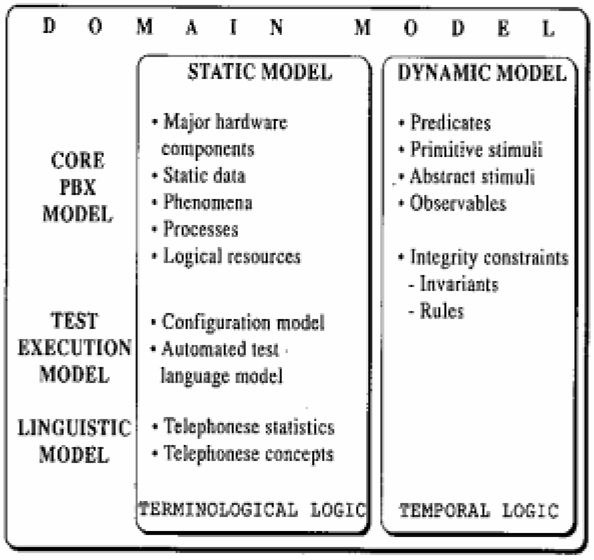

Figure 3: KITSS Domain Model

specifics of the automated test language. This model is used mainly by the translator.
Linguistic model: It is specific to the input language (*telephonese*) and is used mainly by the natural language processor.

From a knowledge representational point of view, we distinguish between static properties of the domain model and dynamic ones [Brodie *et al.*, 1984]. *Static properties* include the objects of a domain, attributes of objects, and relationships between objects. All static parts of the domain model are implemented in a terminological logic (see Section 3.1). *Dynamic properties* include operations on objects, their properties, and the relationships between operations. The applicability of operations is constrained by the attributes of objects. *Integrity constraints* are also included to express the regularities of a domain. The dynamic part of the core PBX model is represented in temporal logic (see Section 3.2).

3.1 Static Model

This part of the domain model represents the static aspects of KITSS. By static we mean all objects, data, and conditions that do not have a temporal extent but may have states or histories.
The **static PBX model** includes the following pieces:

- *Major hardware components*, such as telephones and switch administration consoles as well as smaller subparts of theses components, *e.g.*, buttons, lamps, and handsets.
- *Static data, e.g.*, telephone numbers, routing codes and administrative data such as available features, and current feature settings.
- *Phenomena*, such as tones and flashing patterns which are occurrences at points in time.
- *Processes*, such as static definition of types of calls (*e.g.* voice calls, data calls, priority calls) and types of sessions (*e.g.* calling sessions, feature sessions).
- *Logical resources*, such as lines and trunks required by processes.

The **test execution model** is divided as follows:

- The *configuration model* describes the current test setup, *i.e.*, how many simulated phones and trunk lines are available or which extension numbers belong to which phones/lines, etc. It also contains the dial plan and the default feature assignments.
- The *automated test language model* defines the vocabulary of the test script language.

The **linguistic model** supports two pieces:

- *Telephonese statistics*, which are frequency distributions of syntactic structures, help the natural language processor by disallowing interpretations of phrases and concepts that are possible in English but not likely in *telephonese*.
- *Telephonese concepts* make it easier to paraphrase KITSS' representations for user interactions.

We used CLASSIC [Brachman *et al.*, 1989] to represent the knowledge in our domain. CLASSIC belongs to the class of terminological logics (*e.g.* KL-ONE). It is a frame-based description system that is used to define structured concepts and make assertions about individuals. CLASSIC organizes the concepts and the individuals into a hierarchy by classification and subsumption. Additionally, it permits inheritance and forward-chaining rules. CLASSIC is probably the most expressive terminological logic that is still computationally tractable [Brachman *et al.*, 1990]. Queries to CLASSIC are made by *semantics* not by syntax.

The static model incorporates multiple views of an object from the various models into one (*e.g.*, a station might have one name in the English test case, another in the automated test language and a third in the actual configuration). Thus, although each reasoning module might have a different view on the same object, CLASSIC will always retrieve the same concept correctly.

3.2 Dynamic Model

This unique part of the domain model represents all dynamic aspects of the switch's behavior. It basically defines constraints that have to be fulfilled during testing as well as the predicates they are defined upon.
The **dynamic PBX model** includes the following pieces:

- *Predicates*, such as offhook, station-busy, connected, or on-hold, define a state which currently holds for the switch. The different phases of a call are described with predicates such as requesting-connection, denied-connection, or call-waiting-for-timeout. Each of the predicates has defined *sorts* that relate to objects in the static model. Synonyms (*e.g.*, on-hold is a synonym for call-suspended) are allowed as well.

- *Stimuli* can be either primitive or abstract. Stimuli appear in the action statements of test cases.

  A *primitive stimulus* defines an action being performed by the user (*e.g.*, dials-extension, goes-offhook) or by the switch (*e.g.*, timeout-call). The necessary pre- and postconditions (before and after the stimulus) are also specified. For instance, for a station to be able to go offhook the precondition is that the station is not already offhook and the postcondition is that the station is offhook after the stimulus[4].

  An *abstract stimulus* is not an atomic action but may have pre- and postconditions like a primitive stimulus. However, there are several primitive stimuli necessary to achieve the goal of a single abstract stimulus (*e.g.*, place-call, busy-out-station, or activate-feature). The steps necessary for an abstract stimulus are defined in one or many *abstract stimulus plans*. The abstract stimulus defines the conditions that need to be true for the goal to succeed whereas the abstract stimulus plans describe possible ways of achieving such a goal.

- *Observables* are states that can be verified such as receives-tone, ringing, or status-lamp-state. Observables appear in the verify statements of test cases.

Additionally, the dynamic model includes two different types of **integrity constraints**:

- *Invariants* are assertions that 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. The paraphrases of a few of the invariants are as follows: "Only offhook phones receive tones" or "You only get ringing of any kind when you are alerting" or "A forwarded call always alerts at the forwardee, never at the forwarder" or "You can't be be talking to an on-hold call".

- *Rules* also describe low-level behavior in telephony. These are mainly state transitions in signaling behavior like "A tone must stop whenever another begins" or "Stop dial-tone after dialing an extension" or "An idle phone starts to ring when the first incoming call arrives".

[4]Note the difference between the *state* of being offhook and the *action* goes-offhook.

Representing the dynamic model we required expressive power beyond CLASSIC or terminological logics. For example, CLASSIC is not well-suited for representing plan-like knowledge, such as sequences of actions to achieve a goal, or to perform extensive temporal reasoning [Brachman *et al.*, 1990]. But this is required for the dynamic part of KITSS (see above examples). We therefore used the WATSON Theorem Prover (see Section 4.2), 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* which are sufficient to represent all temporal aspects of our domain. For example, the abstract stimulus plan for activating a feature is represented in temporal logic as follows.

```
(abstract-stimulus-plan activate-feature-1
 ((:plan-goal activate-feature)
  (:sorts
   ((station s1) (feature f) (station s2)))
  (:preconditions
   ((holds (onhook s1))))
  (:plan-steps
   (((occurs (initiate-feature-session s1 f))
     (begins (receives-tone s1
                       second-dial-tone)))
    ((occurs (dials-destination s1 s2))
     (issues (receives-tone s1
                       confirmation-tone)))
    ((occurs (terminate-feature-session s1 f))
)))))
```

The theorem proving is tractable due to the tight integration between knowledge representation and reasoning. Therefore, we specifically designed the analyzer using the WATSON Theorem Prover and targeting them for this domain. The challenging task in building the dynamic model was to understand and extract what the invariants, constraints, and rules were [Zave & Jackson, 1991]. Representing them then in the temporal logic was much easier.

3.3 Domain Model Benefits

In choosing a hybrid representation, 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.

There were other benefits to building a domain model. It ensures that a standard terminology is used by all of the test case authors. The domain model also simplifies the maintenance of test scripts. In automated testing environments without a domain model, the knowledge is scattered throughout thousands of scripts. With the domain model a change in the functioning of the software is made in only one place which makes it possible to centralize knowledge and therefore centralize the maintenance effort. Additionally, the domain model provides the knowledge that reduces and simplifies the tasks of the natural language processor, the analyzer, and the translator modules.

4 Reasoning Modules

4.1 Natural Language Processor

The existing testing methodology used English as the language for test cases (see Figure 1) which is also KITSS' input. Recent research in statistical parsing approaches [Jones & Eisner, 1991] provided some answers to the difficulty of natural language parsing in restricted domains such as testing languages. In the KITSS project, the parser uses probabilities (based on training given by *telephonese* examples) to prune the number of choices in syntactic and semantic structures. Unlikely structures can be ignored or eliminated, which helps to speed up the processing. For instance, consider the syntax of the following two sentences[5]:

Place a call to station troops in Saudi Arabia.
Place a call to station "4623" in two minutes.

Both examples are correct English sentences. Although the second sentence on the surface matches in many parts the first one, their structure is very different. In the first sentence "station" is a verb, in the second a noun; "to" is an infinitive and a preposition respectively. "In Saudi Arabia" refers to a location whereas "in two minutes" refers to time. It is hard to come up with correct parses for both but by restricting ourselves to the *telephonese* sublanguage this is somewhat easier. In *telephonese*, the structure of the first sentence is statistically unlikely and can be ignored while the second sentence is a common phrase.

[5]This example was given by Mark Jones.

The use of statistical likelihoods to limit search during natural language processing was used not only during parsing but also when assigning meaning to sentences, determining the scope of quantifiers, and resolving references. When choices could not be made statistically, the natural language processor could query the domain model, the analyzer, or the human user for disambiguation. The final output of the natural language processor are logical representations of the English sentences, which are passed to the analyzer.

4.2 Completeness & Interaction Analyzer

The completeness and interaction analyzer represents one of the most ambitious aspects of KITSS. It is based on experience with the WATSON research prototype [Kelly & Nonnenmann, 1991]. Originally, WATSON was designed as an automatic programming system to generate executable specifications from episodic descriptions in the telephone switching software domain. This was an extremely ambitious goal and could only be realized in a very limited prototype. To be able to scale up to real-world use, the focus has been shifted to merely checking and augmenting given tests and maybe generating related new ones rather than generating the full specification.

Based on the natural language processor output, the analyzer groups the input logical forms into several *episodes*. Each episode defines a stimulus-response-cycle of the switch, which roughly corresponds to the action/verify statements in the original test case. These episodes are the input for the following analysis phases. Each episode is represented as a logical rule, which is checked against the dynamic model. The analyzer uses first-order resolution theorem proving in a temporal logic as its inference mechanism, the same as WATSON.

The analysis consists of several phases that are specifically targeted for this domain and have to be re-targeted for any different application. All phases use the dynamic model extensively. The purpose of each phase is to yield a more detailed understanding of the original test case. The following are the current analysis phases:

- The structure of a test case is analyzed to recognize or attribute purpose to pieces of the test case. There are four major pieces that might be found: administration of the switch, feature activation or deactivation, feature behavior, and regression testing.

- The test case is searched for connections among concepts, *e.g.*, there might be relations between system administration concepts and system signaling that need to be understood.

- Routine omissions are inserted into the test case. Testers often reduce (purposefully or not) test sequences to their essential aspects. However, these omissions might lead to errors during testing and therefore need to be added.

- Based on the abstract plans in the dynamic model, we can enumerate possible specializations, which yield new test cases from the input example.

- Plausible generalizations are found for objects and actions as a way to abstract tests into classes of tests.

During the analysis phases, the user might interact with the system. We try to exploit the user's ease at verifying or falsifying examples given by the analyzer. At the same time, the initiative of generating the details of a test lies with the system. For example, some test case might violate the *look & feel* of the system, *i.e.*, there is a conflict with an invariant. However, the user might want this behavior intentionally which will lead to a change in the *look & feel* itself.

The final output of the analyzer is a corrected and augmented test case in temporal logic. As an example of the analyzer's representation after analysis, the following shows the logical forms for the first few episodes in Figure 1. Notice that the test case is expanded since the analyzer applied abstract stimulus plans.

```
...
((OCCURS (GOES-OFFHOOK B))
 (BEGINS (RECEIVES-TONE B NORMAL-DIAL-TONE)))
((OCCURS (DIALS-CODE B
           (ACTIVATE-ACCESS-CODE CF)))
 (BEGINS (RECEIVES-TONE B SECOND-DIAL-TONE)))
((OCCURS (DIALS-EXTENSION B C))
 (ISSUES (RECEIVES-TONE B CONFIRMATION-TONE))
 (BEGINS (STATUS-LAMP-STATE B (BUTTON CF)
                            STEADY)))
...
```

This representation is passed to the translator.

4.3 Translator

To make use of the analyzer's formal representation, the translator needs to convert the test case into an executable test language. This language exercises the switch's capabilities by driving test equipment with the goal of finding software failures. One goal of the KITSS project was to extend the life of test cases so that they could be used as many times as possible. To accomplish this, it was decided to make the translator support two types of test case independence.

First, a test case must be test machine independent. Each PBX that we run our tests on has a different configuration. KITSS permits a test author to write a test case without knowing which particular machine it will be run on and assuming unlimited resources. The translator loads the configuration setup of a particular switch into the *test execution model*. It uses this to make the test case concrete with respect to equipment used, system administration performed, and permissions granted. Thus, if the functional description of a test case is identical in two distinct environments, then the logical representation produced by the earlier modules of KITSS should also be identical.

Second, a test case must be independent of the automated test language. KITSS generates test cases in an in-house test language. The translator's code is small because much of the translation information is static and can be represented in CLASSIC. If a new test language replaces the current one then the translator can be readily replaced without loss of test cases, with minimal changes to the KITSS code, and without a rewrite of most of the domain model.

5 Status

The KITSS project is still a prototype system that has not been deployed for general use on the DEFINITY project. It was built by a team of researchers and developers. Currently, it fully translates 38 test cases (417 sentences) into automated test scripts. While this is a small number, these test cases cover a representative range of the core features. Additionally, each test case yields multiple test scripts after conversion through KITSS. The domain model consists of over 500 concepts, over 1,500 individuals, and more than 80 temporal constraints. The domain model will grow somewhat with the number of test cases covered, however, so far the growth has been less than linear for each feature added.

All of the modules that were described in this paper have been implemented but all need further enhancements. System execution speed doesn't seem to be a bottleneck at this point in time. CLASSIC's fast classification algorithm's complexity is less than linear in the size of the domain model. Even the analyzer's theorem prover, which is computationally the most complex part of KITSS, is currently not a bottleneck due to continued specialization of its inference capability. However, it is not clear how long such optimizations can avoid potential intractability.

The current schedule is to expand KITSS to cover a few hundred test cases. To achieve this, we will shift our strategy towards more user interaction. The version of KITSS currently under development will intensely question the user to explain unclear passages of test cases. We will then re-target the reasoning capabilities of KITSS to cover those areas. This rapid-prototyping approach is only feasible since we have already developed a robust core system. Although scaling-up from our prototype to a real-world system remains a hard task, KITSS demonstrates that our knowledge-based approach chosen for functional software testing is feasible.

6 Conclusion

As we have shown, testing is perhaps one of the most expensive and time-consuming steps in product design, development, and maintenance. KITSS uses some novel approaches to achieving several desirable goals. Features will continue to be specified in English. To support this we have incorporated a statistical parser that is linked to the domain model as well as to the analyzer. Additionally, KITSS will interactively give the user feedback on the test cases written and will convert them to a formal representation. To achieve this, we needed to augment the domain model represented in a terminological logic with a dynamic model written in a temporal logic. The temporal logic inference mechanism is customized for the domain. Tests will continue to be specified independent of the test equipment and test environment and the user will not have to provide unnecessary details.

Such a testing system as demonstrated in KITSS will ensure project-wide consistent use of terminology and will allow simple, informal tests to be expanded to formal and complete test scripts. The result is a better testing process with more test automation and reduced maintenance cost.

Acknowledgments

Many thanks go to 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

[Balzer *et al.*, 1977] Balzer R., Goldman N., and Wile D.: Informality in program specifications. In *Proceedings of the 5th IJCAI*, Cambridge, MA, 1977.

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

[Brachman *et al.*, 1989] Brachman, R.J., Borgida, A., McGuinness, D.L., and Alperin Resnick, L.: The CLASSIC knowledge representation system, or, KL-ONE: The next generation. In preprints of *Workshop on Formal Aspects of Semantic Networks*, Santa Catalina Island, CA, 1989.

[Brachman *et al.*, 1990] 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.

[Brodie *et al.*, 1984] Brodie, M.L., Mylopoulos, J., and Schmidt, J.W.: *On conceptual modeling: Perspectives from Artificial Intelligence.* Springer Verlag, New York, NY, 1984.

[Brooks, 1987] Brooks, F.P.: No silver bullet: Essence and accidents of software engineering. *Computer*, Vol. 20, No. 4, April 1987.

[Howden, 1985] Howden, W.E.: The theory and practice of functional testing. *IEEE Software*, September 1985.

[Jones & Eisner, 1991] Jones, M.A., and Eisner J.: A probabilistic chart-parsing algorithm for context-free grammars. *AT&T Bell Laboratories Technical Report*, 1991.

[Kelly & Nonnenmann, 1991] 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.

[McCartney, 1991] McCartney, R.: Knowledge-based software engineering: Where we are and where we are going. In *Automating Software Design*, M. Lowry and R. McCartney, eds., MIT Press, 1991.

[Myers, 1976] Myers, G.J.: *Software Reliability.* John Wiley & Sons, New York, NY, 1976.

[Myers, 1979] Myers, G.J.: *The Art of Software Testing.* John Wiley & Sons, Inc. New York, NY, 1979.

[Nonnenmann & Eddy, 1991] 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.

[Yonezaki, 1989] Yonezaki, N.: Natural language interface for requirements specification. In *Japanese Perspectives In Software Engineering*, Y. Matsumoto and Y. Ohno, eds., Addison-Wesley, 1989.

[Zave & Jackson, 1991] Zave, P, and Jackson, M.: Techniques for partial specification and specification of switching systems. In *Proceedings of the VDM'91 Symposium*, October 1991.