Applying Formal Methods to a Telecommunications System in a Commercial Setting

Andre Wong Marsha Chechik *

October 19, 1998

Abstract

Formal methods have long been advocated by the software engineering research community; however, most successful applications of formal methods are confined to safety-critical projects where software correctness is the pivotal goal. In contrast, the software industry seeks practical techniques that can be seamlessly integrated into their existing processes and improve productivity; absolute quality is often a desirable but not crucial objective. A number of formal methods have been designed to address this issue, and some have industrial-strength implementations that can be used effectively in commercial settings.

This paper describes a case study conducted in collaboration with Nortel to demonstrate the feasibility of applying formal methods to telecommunications systems. A lightweight formal method, SDL, was chosen by our qualitative CASE tool evaluation to model a multimedia-messaging system described by an 80-page natural language specification. Our model was used to identify errors in the software requirements document and to derive test suites, shadowing the existing development process and keeping track of a variety of productivity data.

Creating a model which was easy to review by Nortel engineers, allowed us to locate specification errors that were missed by several manual inspections. In addition, the model was used to derive a variety of test cases that doubled the number of errors discovered by Nortel testers. The success of our case study was in finding a suitable project where we were able to integrate a well-chosen formal method into the existing development process.

*Contact address: Department of Computer Science, University of Toronto, Toronto, ON M5S 3G4, Canada. Email: {andre,chechik}@cs.toronto.edu.

1 Introduction

For a long time, researchers and practitioners have been seeking different ways to improve productivity in the software development process. Within the software engineering research community, formal methods (mathematically-based languages, techniques or tools for specifying and verifying software systems [4]) have been advocated as one of the viable approaches. If high quality specifications are crucial to the success of system developments, it seems logical to apply formal specification techniques to the requirements for ensuring their completeness and consistency.

However, most successful applications of formal methods have been confined to safety-critical projects [6, 9, 13] where software correctness is the pivotal goal. In contrast, the software industry seeks practical techniques that can be seamlessly integrated into their existing processes and improve productivity; absolute quality is often a desirable but not crucial objective. Although the feasibility of some formal methods have been demonstrated in commercial settings [8, 10, 15], the overall adoption has been slow, perhaps because of myths [3, 7] such as the deep mathematical background and the substantial initial investment needed. Most companies, such as the Canadian-based telecommunications company Nortel[1], opt to rely on manual inspections of natural-language specifications as the only requirement error detection technique, even though the results have been suboptimal. If the advantages of better quality specifications, such as clearer understandings of the systems and less error-prone designs, alone do not provide adequate justifications, more benefits can be obtained by leveraging the investment in the formalization process to other stages of the software lifecycle, i.e. generating code or test cases from the formal specifications. Not only does this amortize the cost of creating the specifications, but the productivity improvement will also be easily measurable and more immediate.

Driven by the need to improve the software development phase, Nortel and the Formal Methods Laboratory at the University of Toronto have jointly proposed a research project to investigate the feasibility of formal methods in a commercial setting. The goal of the project is to find means of using formal methods to improve productivity in various stages of the software lifecycle. Specifically, the emphasis is placed on deriving test cases from the formal model as the Nortel engineers have expressed concerns about the feasibility of code generation for their proprietary platform.

The project was organized as a hybrid quantitative and qualitative case study [14]. As it was extremely important to choose the right system/formal method combination for the formalization process, we began the study by proposing criteria for the system selection process and conducting a qualitative formal method evaluation. Our model was used to identify errors in the software requirements document and to derive test suites, shadowing the existing development process. Throughout the study, a variety of productivity data were kept for comparison with similar information from the actual development process.

[1]Nortel, for the purpose of this paper, refers to the Toronto Multimedia Applications Center of Nortel Networks.

The qualitative impacts of the formalization process were also noted.

The rest of the paper is organized as follows: Section 2 provides a brief description of the software system modeled in the study and explains the rationale behind its selection. Section 3 discusses the criteria used in selecting a lightweight formal method suitable for the study. In Section 4, we discuss the formalization process. Section 5 presents findings from the study, and Section 6 draws conclusions based on the experience gained during this project.

2 System Selection

Choosing the appropriate system guarantees half of the success of the study. As not all systems are worth engineering [16], only some systems are worth formalizing. For instance, there can only be limited benefits in formalizing a GUI system when simpler methods such as prototyping are better at handling systems with frequently changing requirements. Therefore, it was crucial to select a system that had the best potential to benefit from the formalization process. We felt that the impact of our study would be more apparent if the system to be modeled

- had relatively stable requirements;
- was not too broad in scope but contained complex parts;
- was relatively self-contained and had loose coupling with the underlying system;
- was reactive—almost all inputs resulted in outputs that accurately indicated the state of the system [17].

We worked with a group of Nortel engineers, consisting of developers from the design team and testers from the verification team, to choose a system that satisfied the criteria above. The software chosen was a subsystem in the Operation, Administration and Maintenance (OAM) software of a multimedia-messaging system connected to a private branch exchange. The subsystem, called Service Creator in our paper, is a voice service creation environment that lets administrators build custom telephony applications in a graphical workspace by connecting predefined voice-service components together. Figure 1 illustrates the graphical view of one such telephony application consisting of four components: the *start*, *end*, *password-check*, and *fax* components. When the application is activated, a call session begins at the *start* component, and a caller is required to enter a numerical password in order to retrieve a fax document from the *fax* component. The caller is directed to the *end* component if an incorrect password is entered. In both scenarios, the call session ends when the *end* component is reached.

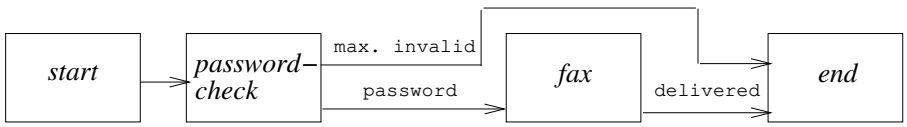


Figure 1: A simplified telephony application example.

The lines connecting various components represent potential control flows of the call session, and the actions performed by the caller in an active component determine the output path taken. In the *password-check* component, for instance, the caller exits via the path *password* if a correct password is entered, or the path *max. invalid* if there are too many invalid password attempts.

In our study, we analyzed the run-time behavior of 15 such components, described by an 80-page natural language specification. However, due to space limitations, we focus our discussion to the analysis of the password-check component, described by a 5-page natural language specification. Figure 2 shows a graphical view of this component.

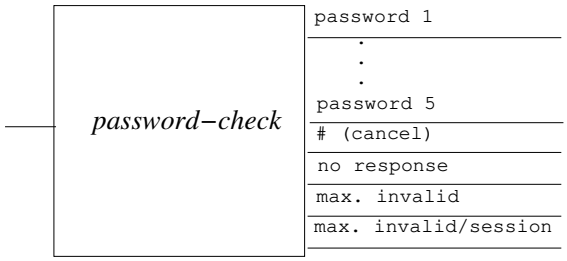


Figure 2: The graphical view of the password-check component.

The purpose of this component is to validate digits entered by a caller against any of the passwords (up to five) defined by the administrator. For instance, the path *password 1* is taken if the entered digits match the first defined password. The caller is forced to leave the component using the *max. invalid* output if the maximum number of invalid password attempts is reached. Such attempts are also monitored on a per-call session basis, and the caller leaves via the *max. invalid/session* output if the per-call session limit is reached. The caller can also enter the * key to retrieve the help prompt, which has a side effect of clearing the password entry, or the # key to exit prematurely via the *# (cancel)* path if no password has been entered. If the caller stays idle for a certain time period and has not previously keyed in any digits prior to entering the password-check component, she is assumed to be using a rotary phone and is transferred to a different voice-service component. Otherwise, one of the two delay prompts may be played depending on whether she has begun keying in the password. After two more timeouts, the caller exits via the *no response* path.

3 Formal Method Evaluation

The goal of the evaluation was to select a suitable formal method to be used in the feasibility study. The success of our project depended as much on the choice of the formal method as on the system selection. Not only did a suitable formal method ease the difficulty of the formalization process, but it also had a direct impact on the results of the study.

Clearly, easily readable and reviewable artifacts as well as a simple notation were two basic requirements for a formal method to be usable in a commercial setting. Moreover, since one of the overall objectives was to amortize the cost of creating a formal specification, we began the evaluation by conducting a broad survey [20] of usable formal methods that provided support both for modeling and for testing. These constraints proved to be extremely limiting as most of the tools surveyed either concentrated in only one of these areas, did not support the use of formal notation, or were simply too difficult to be used in industry. We eventually narrowed down our search to the following candidates:

- *Telelogic SDT* [18]—an integrated software modeling and testing tool suite that utilizes Specification and Description Language (SDL) [11], which is based on extended finite state machines (EFSMs), for behavioral modeling and Message Sequence Charts (MSCs) [12] for component-interaction specification. MSCs, which can be used as test cases, can be derived semi-automatically from an SDL model. Alternatively, SDT can verify the correctness of the model with respect to independently created MSCs.
- *Aonix Validator/Req (V/Q)* [2]—a test generation tool that allows the black-box requirements of a system to be specified in the form of parameterized UML [5] use cases. Validator/Req generates test cases for data-flow testing from the model automatically and produces documents that conform with the IEEE standard for software requirements specifications [1].
- *Teradyne TestMaster (TM)* [19]—a test generation tool that automatically generates test cases for both control-flow and limited data-flow testing from models based on EFSMs. The number of test cases can be flexibly tuned to suit different testing needs.

To perform a detailed assessment, we structured our evaluation more formally as a qualitative case study [14] and refined our focus to choosing among the remaining tools using additional evaluation criteria gathered from the Nortel engineers. These criteria comprised of factors, such as usability and smooth integrations, that were crucial to the use of formal methods in their environment. After the tools were used to model the password-check component, they were ranked against the criteria based on our impressions of the tools and the models produced.

Table 1 shows results of this evaluation. It lists the evaluation criteria in column one, their relative importance using a scale from 1 (least important) to 5 (most important) in

Criteria	Weighting	SDT	V/Q	TM
has modeling support	5	4	1	3
has testing support	5	3	4	5
has a gentle learning curve	4	3	5	3
produces easy-to-understand artifacts	3	3	5	4
enhances understanding of the system	4	5	3	4
integrates smoothly with existing development process and tools	4	5	3	3
is scalable	3	4	2	3
has strong industrial support	3	5	4	5
performs completeness or consistency checks	2	3	3	3
provides features such multi-user support, requirements traceability and document generation	1	3	4	2
Score	N/A	**132**	**112**	**124**

Table 1: Formal Methods Comparison

column two[2], and the degree the tools satisfy the criteria using a scale from 1 (unsatisfactory) to 5 (complete satisfactory) in columns three to five. In the first row, for instance, SDL scores the highest as it allows the behavioral modeling and the hierarchical partitioning of a system into concurrent processes. TestMaster has similar modeling support but a system can only be modeled as a single hierarchical EFSM. Validator/Req ranks the lowest as it provides no modeling support. The conclusion of the evaluation was reached by comparing final scores of the tools which were obtained by summing their weighted scores from each criterion, making SDT the most suitable tool.

A confounding factor in qualitative case studies is the potentially biased opinions from the evaluator. Although we tried to ensure the objectivity of the evaluation, the assignments of scores inevitably contained our subjective judgment, and we did not feel that we could accurately evaluate factors such as usability. To mitigate this potential problem and to gain more confidence in our assessment, we demonstrated the tools and the models to the engineers. They agreed that SDT met their proposed criteria more closely than the other tools.

4 Modeling the System

The next step was to formalize a Service Creator application in parallel with the actual development process. The system was modeled as a 70-page SDL system in which the environment contained the underlying OAM software and messaging-system hardware. Of the 15 voice-service components, 10 were modeled as separate SDL processes that communicated with the environment through a "driver" SDL process (see Appendix A for an example), which was responsible for administrating the control flow of the telephony

[2]These were assigned after consultations with Nortel engineers.

application being modeled, i.e. activate the appropriate component when a caller exits via an output path. The functionality of the five remaining components (e.g., the *start* and *end* components) were incorporated into the "driver" processes. A total of 23 signals were used in the SDL system; eight of them were external (used for communicating with the environment) and the rest internal. Persistent data such as the predefined passwords for the password-check component were represented as parameters to the SDL processes.

4.1 The Level of Abstraction

The modeling process was relatively straightforward as we encountered no major problems in modeling the Service Creator; we also felt that a background in formal methods was not required. However, the biggest concern was to determine the appropriate level of abstraction which was dictated by two opposing needs: the model should be constructed from a black-box view to reduce its complexity, while the exact behavior of the system might need to be modeled for deriving test cases. In addition, a more detailed model would help in identifying problems in the natural-language specification.

Our approach was to start from a high level of abstraction, filling in details about some parts of the behavior if the natural-language specification required it. For example, in modeling the password-check component, we represented the various timeouts by an SDL timer *timeout* (see Figure 3), as the actual lengths of the timeouts were relatively unimportant. Processing of user input, on the other hand, required reasoning on the level of single digits. In our model, a received digit, *digit*, was actually stored in a variable *numberRecv*. While this treatment could lead to large and cluttered models, we sometimes had to resort to this level of abstraction to be able to derive sufficiently detailed test cases.

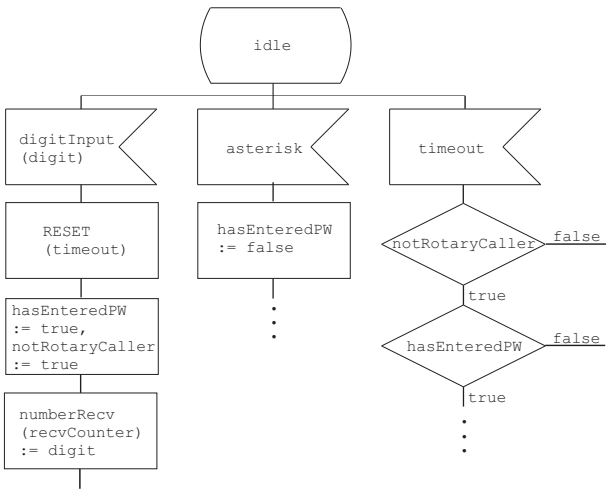


Figure 3: A simplified fragment from the SDL model of the password-check component.

4.2 Test Case Derivation

To obtain immediate benefits from the formalization process, 120 MSCs were derived from the SDL model for testing the implementation. The derivation was not automatic: SDT recorded the interactions between the SDL system and its environment as MSCs when we animated the SDL model manually. However, we felt that the automation was not necessarily desirable since this exercise also gave us more assurance and confidence in the content of the test cases.

During the test case derivation, we took advantage of the modular nature of the voice-service components and generated test cases for each of them separately, achieving full transition coverage in their corresponding SDL processes. However, from the description of the password-check component, it might be clear that some functionality could only be covered by testing multiple components together. For instance, to derive test cases for testing the initial timeout behavior for touch-tone callers where no passwords were keyed in, we created a telephony application model in which a caller was required to key in some digits in a component, such as a menu, prior to entering the password-check component. Only by linking them together could we test this specific timeout behavior (see Appendix B for the corresponding MSC). More than 20 test cases of this type were created. In fact, most of the time in this phase was spent on identifying relationships between the components and ensuring that all such relationships were considered.

4.3 Specification and Implementation Problems

The SDL model and the experience gained from the formalization process allowed us to identify specification errors that escaped a series of manual inspections by Nortel engineers. As the components modeled were not particularly complicated, most of the errors we found were caused by vagueness and missing information. In fact, the most time-consuming part of the formalization process was to understand the natural-language specification and to consult the engineers for clarifications. We estimated that these activities alone took as much time as the formalization process itself. Some of the problems found in the specification of the password-check component are presented below:

1) it was never mentioned whether the various voice prompts were interruptible, and no conventions for such behavior were defined;
2) lengths of various timeouts were not mentioned;
3) it was unclear which exit path should be taken when the administrator set two or more passwords to be the same;
4) the maximum and minimum lengths of passwords were not defined.

Not all problems identified in the study, such as the first problem listed above, were crucial to the correct operation of the system as it was conceivable that some of them could be easily resolved in the implementation. However, as Nortel test cases were developed in

parallel with the implementation, the missing information could hinder the testers' ability to create accurate test cases in a timely manner. This might have an effect on the length of the testing phase and possibly the overall development.

The MSCs derived from the SDL model were used to identify errors in the implementation after Nortel engineers officially completed testing of the voice-service components. It was found that some of the specification errors, such as the fourth problem listed above, actually propagated to the implementation, i.e. the graphical interface for defining passwords were unable to handle long passwords. In addition, even if a short password was setup, a malicious caller could still crash the system by entering an abnormally long password within the password-check component. This was a concrete example of a requirement omission becoming a critical implementation error.

5 Findings

We began our analysis by seeking quantitative evidence to measure the effects of the formalization process on productivity. However, as the study progressed, we felt that it was crucial to identify the qualitative factors (e.g, usability and commitments from the development team) and the limitations of the study in order to reach accurate and unbiased conclusions.

5.1 Quantitative Results

The entire modeling process, which consisted of activities such as understanding and formalizing the specification as well as deriving and executing test cases, took about two person-months to complete. During this period, we kept track of a variety of productivity data in the study (column two of Table 2) for comparison with similar information from the existing development process (column three).

Productivity Data	**Study**	**Existing Process**
Time to model (person-days)	11	N/A
Number of specification errors reported	56	N/A
Number of test units	269	96
Test unit creation rate (test units/day)	30	21
Test unit execution rate (test units/day)	27	11
Number of implementation errors identified	50	23

Table 2: Productivity Comparison.

Since the sizes of test cases varied greatly, we did not use a "test case" as the unit of comparison for testing-related data. Instead, we counted test units, the smallest externally visible functionality of the system, in each test case to ensure a fair comparison. Highlights from the table are summarized below:

- The *time to model* value included only the time used for modeling the SDL processes. It was impossible to draw conclusions from it, since the modeling task did not have an equivalent in the actual development process. However, we felt that the 11 person-days was a fairly small amount of time when compared to the four person-months it took to develop the corresponding specification.
- The number of specification errors reported could not be used for comparison as the Nortel engineers did not keep track of such statistics.
- More test units could be derived from the SDL model (which translated to better test coverage) at a faster rate, possibly because the model eased the creation task by providing a more in-depth understanding of the system as well as a better sense of completeness. One other reason for the difference in the quantity is that the test units from Nortel were sometimes vaguely specified; the missing details contributed to a decrease in the number of test units reported.
- The rate of the test unit execution in our study was much higher for two major reasons: the derived test units were more detailed and thus easier to execute, and it was observed that Nortel engineers spent extra time creating more scenarios based on the vaguely specified test units during the testing phase. These extra test units were usually not recorded in the test-unit database, which also contributed to a decrease in the reported *number of test units* and the *test unit execution rate* values.
- Of the 50 implementation errors identified, 18 were avoidable errors that could be linked to problems in the specification. Moreover, the number of implementation errors identified in the study was two times larger than that of the existing development process, possibly due to better test coverage.

5.2 Qualitative Observations

While all the quantitative data was in favor of the use of the formal modeling, it was clear that these results alone constituted only a part of our findings. Some observations from the formalization process that were not evident from Table 2 are discussed below.

- The most frequent complaint from test engineers is that the missing information in the specifications often complicates the task of test case creation. The SDL models encourage and assist developers in stating system requirements more precisely and completely, which should allow testers to create better quality (e.g., more detailed and with expected results more clearly defined) test cases and reduce the time needed for test case creation and execution. Developers should also benefit from the more complete specifications in the design and implementation phases. This is one area where SDL can potentially improve the development process.
- SDL can be integrated into the existing development process without significant changes. While developers can use SDL models to complement the natural-language specifications by elaborating complicated details, testers can derive test cases from the same model for greater cost-effectiveness. An added benefit is that for parts of

the system that are not formalized, the existing development process can be used without any changes.

- A common problem with most formal methods is that they require requirements to be specified and reasoned about in formal logic—skills that most Nortel engineers do not possess. SDL reduces the problem to possibly a minimum by only requiring them to construct an EFSM using a graphical and user-friendly notation. Since it is still difficult for us to assess if SDL addresses the usability issue adequately, we plan to conduct further studies in this particular area.
- As with any other formal specification techniques, a successful integration of SDL with the development process requires firm commitments from the entire development team. For instance, the developers must ensure that the SDL model is always kept consistent with the system requirements and the code, e.g. last-minute changes in the design and implementation are propagated back to the model. Testers also need to ensure that their test cases always reflect the model accurately. While this is possibly one of the biggest obstacles in applying a formal modeling technique, the advantages provided by SDL should justify the extra effort.

5.3 Limitations

Although we believe that our findings, as a whole, demonstrated the usefulness of SDL in formalizing the Service Creator, we cannot infer that all of our results will apply directly to the actual development process. For instance, results from case studies are usually not generalizable [21]. In our study, both the formal method and the system to be modeled were carefully chosen to maximize the impact of the formalization process. We might not obtain the same results if a different formal method was applied to some other type of systems. Also, since the study was performed in parallel with the actual development process without the active involvement from the Nortel engineers, many activities of the study could not be used for comparison, which made some of the analyses difficult or impossible, e.g. the number of specification errors identified in the formalization process could not be compared because there was no equivalent in the existing development process.

Another limitation arose from the differences in the settings. Since one of the goals of our study was to demonstrate the feasibility of SDL, we took the time to create high quality models and detailed test cases with extreme care. However, under the pressure of the actual development process, the Nortel engineers might not devote the same amount of attention to the formalization process, which might prevent them from obtaining results similar to the study. Furthermore, our prior experience in formal modeling might have also reduced the time and difficulty of the formalization process. While we felt that the process was straightforward, our results might not apply directly because most Nortel engineers had no exposure to formal methods. Yet, we believe that with appropriate training and having an SDL expert on premise, these problems could be overcome.

6 Conclusions and Future Work

In this case study, we formalized the behavior of a multimedia-messaging system in a commercial setting. While we identified quantitative and qualitative improvements of the formalization process over the existing development methodology, the success of the study was also in finding a suitable system and a well-chosen formal method with which requirements could be formalized effectively and economically. Among the outstanding issues, usability was an important one that had not been fully addressed. To reinforce our confidence in the results of our study, we plan to conduct further experiments with the Nortel engineers to investigate if the chosen formal method could be applied by them to obtain comparably good results.

7 Acknowledgment

The authors would like to thank Albert Loe, Steve Okun, Avinash Persaud, and Shawn Turner of Nortel for their technical assistance and continual support throughout the study. The study was supported in part by Nortel and Ontario Graduate Scholarship (OGS).

References

[1] ANSI/IEEE. "IEEE 830: IEEE Recommended Practice for Software Requirements Specifications ". *IEEE*, 1993.

[2] Aonix. "Aonix Home Page". `http://www.aonix.com`, September 1998.

[3] Jonathan Bowen and Michael Hinchey. "Seven More Myths of Formal Methods". *IEEE Software*, 12(4):34–41, July 1995.

[4] E.M. Clarke and J. Wing. "Formal Methods: State of the Art and Future Directions". *ACM Computing Surveys*, 28(4):626–643, December 1996.

[5] UML Partners Consortium. "UML Proposal Document Set". *OMG documents ad/97-08-{02,03,04,05,06,07,08,09}*, September 1997.

[6] J. Crow and B.L. Di Vito. "Formalizing Space Shuttle Software Requirements". In *Workshop Formal Methods in Software Practice*, San Diego, California, January 1996.

[7] Anthony Hall. "Seven Myths of Formal Methods". *IEEE Software*, 7(5):11–19, September 1990.

[8] Anthony Hall. "Using Formal Methods to Develop an ATC Information System". *IEEE Software*, 13(2), March 1996.

[9] Mats P.E. Heimdahl. "Lessons from the Analysis of TCAS II". In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'96)*, San Diego, CA, January 1996.

[10] Jonathan P. Hoare. *"Application of the B-Method to CICS"*, pages 97–124. Prentice Hall International Series in Computer Science, 1995.

[11] ITU-T. "ITU-T Recommendation Z.100: Specification and Description Language (SDL)". *ITU-T*, 1993.

[12] ITU-T. "ITU-T Recommendation Z.120: Message Sequence Chart (MSC)". *ITU-T*, 1993.

[13] Paul K. Joannou. "Experiences from Application of Digital Systems in Nuclear Power Plants". In *Proceedings of the Digital Systems Reliability and Nuclear Safety Workshop*, Rockville, Maryland, 1993.

[14] Barbara Ann Kitchenham. "Evaluating Software Engineering Methods and Tools. Part 1". *ACM SIGSOFT Software Engineering Notes*, 21(1):11–15, January 1996.

[15] Peter Mataga and Pamela Zave. *"Multiparadigm Specification of an AT&T Switching System"*, pages 375–398. Prentice Hall International Series in Computer Science, 1995.

[16] Roger S. Pressman. "Can Internet-Based Applications be Engineered?". *IEEE Software*, 15(5), September 1998.

[17] Tony Savor and Rudolph E. Seviora. "Toward Automatic Detection of Software Failures". *IEEE Computer*, 31(8):68–74, August 1998.

[18] Telelogic. "Telelogic SDT Home Page". `http://www.telelogic.com/solution/tools/sdt.asp`, September 1998.

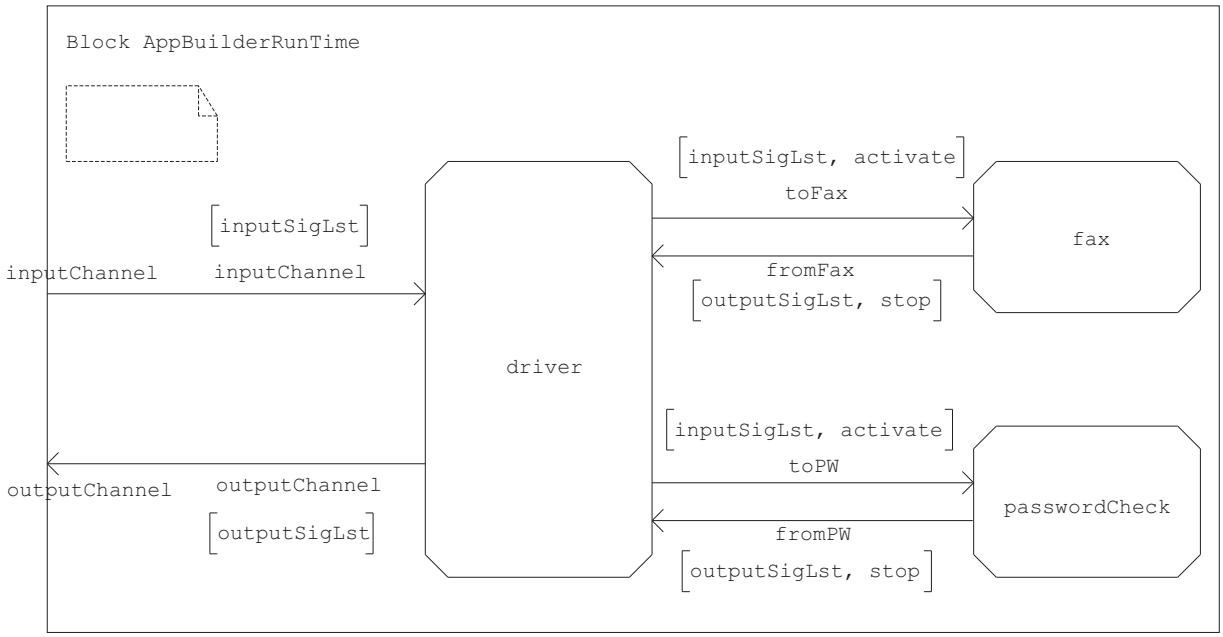
[19] Teradyne. "TestMaster Home Page". `http://www.teradyne.com/prods/sst/ssthome.html`, September 1998.

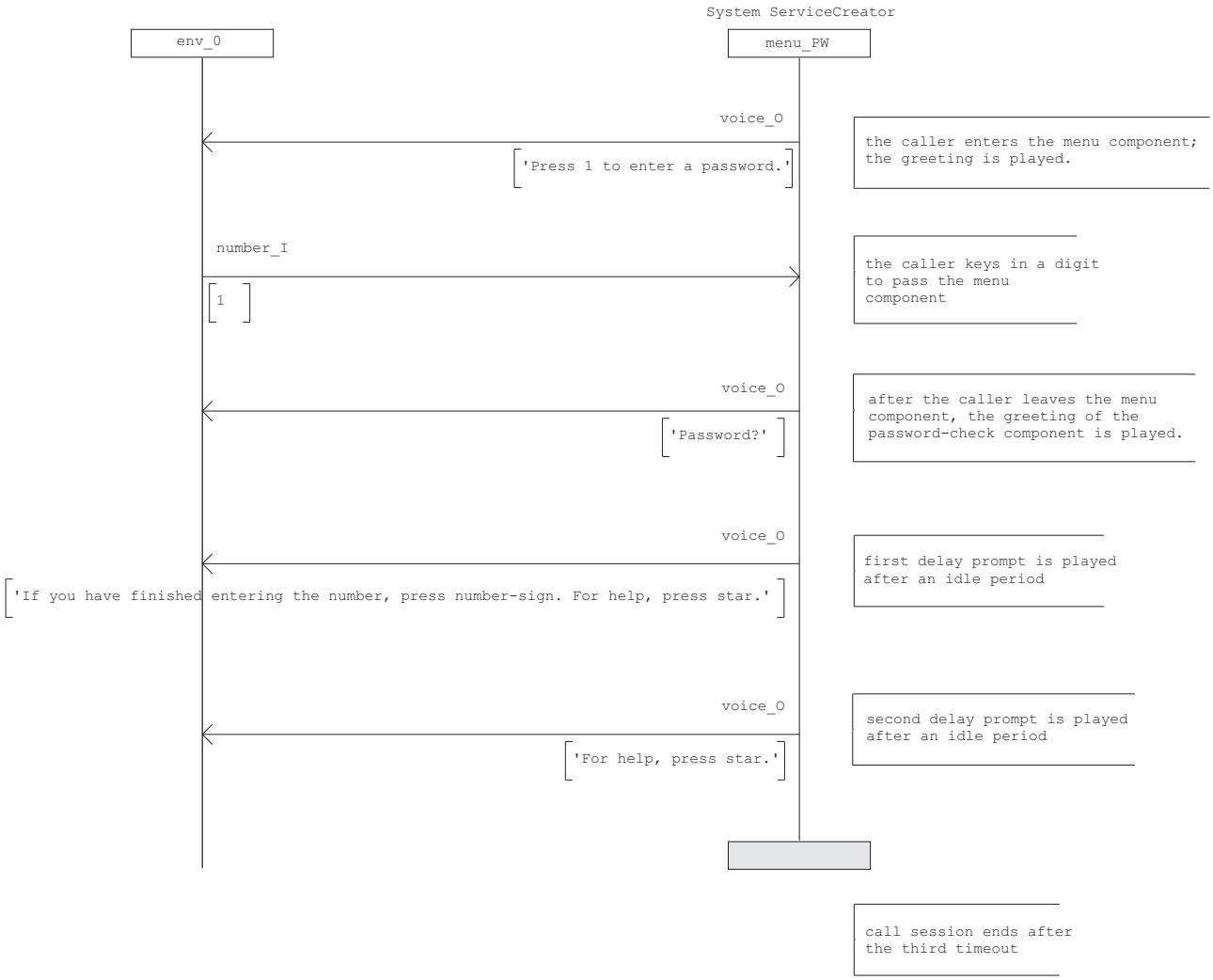
[20] Andre Wong. "Diary of the Formal-Method Survey". `http://www.cs.toronto.edu/~andre/progress.html`, September 1998.

[21] Marvin V. Zelkowitz and Dolores R. Wallace. "Experimental Models for Validating Technology". *IEEE Computer*, 31(5), May 1998.

A An SDL Block Diagram

The diagram below illustrates the SDL block diagram of the telephony application shown in Figure 1. The "driver" block (which contains a "driver" process) acts as the signal router between the environment and the SDL blocks of the voice-service components by routing the signal lists *inputSigLst* and *outputSigLst*. It is also responsible for activating the appropriate component according to the control-flow of the application and actions from the caller by using the signal *activate*. Please refer to Section 4 for more details.



B A Message Sequence Chart

The MSC below shows the interactions between a caller and a telephony application created in the Service Creator. The application requires a caller to press button *1* in the menu component prior to entering the password-check component. Please refer to Section 4.2 for more details.