Formal Modeling in a Commercial Setting: A Case Study

Andre Wong and Marsha Chechik

Department of Computer Science, University of Toronto,
Toronto, ON M5S 3G4, Canada,
{andre,chechik}@cs.toronto.edu,
WWW home page: http://www.cs.toronto.edu/~chechik

Abstract. Formal modeling, particularly of software requirements, has long been advocated by the software engineering research community. Usually, such modeling is linked with formal verification and is 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 the existing processes and improve productivity; very high quality is often a desirable but not crucial objective. A number of modeling languages have been designed to address this issue, and some have industrial-strength tool support that can be used effectively in commercial settings.

This paper describes a case study conducted in collaboration with Nortel Networks to demonstrate the *economic* feasibility and effectiveness of applying formal modeling techniques to telecommunication systems. A formal description language, SDL, was chosen by a CASE tool evaluation study 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.

Our results clearly show that it is possible to use formal modeling in the commercial setting effectively: we were able to locate a number of specification errors that were missed by several manual inspections, and used the model to derive test cases that doubled the number of errors discovered by Nortel testers. The formalization did not lengthen the overall time to market of this product.

1 Introduction

The commercial software industry has been facing a software development dilemma: the public expects (or soon will expect) software systems to have the same high quality as found in other engineering artifacts, while the competitive market pressures the industry to place less emphasis on quality and concentrate on delivering products in the shortest possible time frame. A lot of development techniques used in industry are inadequate for building systems quickly and economically, leading too often to projects that are either late, over budget, or deliver systems that do not meet requirements of their customers [22].

Specifying the system precisely, i.e., using formal description techniques (FDTs), has been advocated as a viable approach to improving both the productivity and the quality of software products [40]. In particular, it can lead to the following benefits [11, 29]:

- The process of formalizing specifications provides a more detailed understanding of the systems by encouraging software engineers to analyze requirements in depth, raising issues that may not otherwise manifest themselves until later in the development;
- Applying FDTs exposes omissions, unstated assumptions and ambiguity that are commonly found in natural-language specifications;
- FDTs help detect requirements errors early in the requirements lifecycle, providing a potential to reduce the length of the overall development and the cost of fixing the errors;
- More precise and cleaner requirements help achieve lower code complexity [41].

FDTs developed into an active research area, resulting in a wide variety of methods, many of which have been successfully applied to specifying industrial-size systems [13, 15, 3, 34, 10, 43]. However, the majority of applications have been confined to the safety-critical domain [16, 26, 33], although such systems account for only a small fraction of the software industry. The reason has been attributed to the gap between what the academia has to offer and what the commercial software industry needs [2, 7, 37, 50]. In particular, a large majority of formal methods address only the improvement in software quality rather than the overall productivity, and thus are perceived to be useful only in the realm of safety-critical systems.

Profit-driven software companies, on the other hand, concentrate on attaining shorter time-to-market, reduced costs and higher productivity [28], and thus place a greater emphasis on obtaining immediate and visible rather than long-term benefits. Often, "zero defects" software quality is a desirable but not crucial objective, and is preferable only if it improves the cost-effectiveness of the overall development process. Since the majority of formal methods do not address practical issues such as usability and maintainability and require access to experts with strong mathematical background, software industry simply do not consider FDTs as a practical alternative to the established processes [12, 48].

The claims about the recently improved usability and cost-effectiveness of formal methods are still largely unsubstantiated, and this lack of evidence has been blamed for the slow adoption of FDTs [47]. Without substantial evidence of the promised benefits from relevant case studies, it is inconceivable that the industry will adapt a relatively radical technique, especially since almost all of such techniques lengthen the requirements analysis phase. We need a large number of relevant case studies in different application domains to substantiate the claims and provide practitioners with real examples on which to base their own pilot studies. In particular, measured productivity gains affecting the entire software lifecycle are best achieved by applying FDTs in the following manner:

- The formalization process should be *lightweight* [32, 17], i.e., allow for creating partial specifications, "without a commitment to developing complete, consistent formal specifications." [17]. This also includes employing easy-to-use formal notations that are suitable for the target application domain.
- The investments made in the formalization process should be leveraged to other stages of the software lifecycle to obtain immediate and visible benefits. For example, this may include generating code or test cases from formal specifications. Not only does this amortize the cost of creating the specifications, but the productivity improvements can also be more immediate and more easily measurable.
- FDTs should be applied to real commercial systems that are being developed and maintained over time to address issues such as scalability and maintainability;

Although carefully-controlled studies are essential in establishing the feasibility and benefits of FDTs, we have to first start with pilot studies. This paper describes a case study conducted in collaboration with the Canadian-based telecommunications company Nortel[1] to demonstrate the *economic* feasibility and effectiveness of applying formal modeling techniques to telecommunication systems. All products developed by the group that we worked with were not considered to be safety-critical or high-availability.

A formal description language, SDL, was chosen by a CASE tool evaluation study 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. In this paper, we concentrate on the modeling and testing aspects of the project, showing in detail the kinds of errors that can be discovered using formal modeling. The major contribution of this work is in its emphasis on highlighting and empirically quantifying the impact of formalization on the productivity improvement.

The rest of this paper is organized as follows: Section 2 provides a more detailed description of the overall project and introduces the software system selected for the study. Section 3 discusses the criteria for choosing a suitable modeling language and briefly describes the Specification and Description Language (SDL). In Sections 4 and 5, we discuss the formalization process and the derivation of test cases from the model, respectively. Section 6 presents findings from the study. This section also summarizes the experience gained during the project and discusses the limitations of this work. Section 7 concludes the paper with the summary of related work and an outline of future research directions.

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

2 Project Description and System Selection

The main object of our project was to determine the effectiveness and quantify the costs/benefits of applying FDTs to commercial systems over the traditional development techniques. In order to do so, we needed to explore the technical and the organizational issues in applying FDTs and integrate them into the existing development process. The main emphasis was placed on being able to *derive test cases* from the formal model.

2.1 Project Description

It is crucial to explore the problems in detail before proposing any solution. In the first phase of the project, we conducted an exploratory study with the Nortel engineers to identify the major problems in Nortel development process, and investigate the causes of these problems and their implications on productivity. The results allowed us to pinpoint the major areas for improvement and determine if FDTs represented a viable solution.

In the second part of the project, we conducted a formal specification pilot study to exemplify the feasibility of FDTs in a commercial setting. Together with the Nortel engineers, we chose a software system that was representative of typical projects in Nortel and had a good potential to benefit from the study. Using the results from the exploratory study and feedback from the engineers, we conducted a survey to select an FDT that tackled the identified problems most effectively and best met the Nortel engineers' expectations of a usable specification technique. The case study chose SDL, supported by a commercial tool SDT [46]. Applying SDL in a lightweight manner, we then specified crucial components of the selected telecommunications system and derived test cases from the resulting formal model, shadowing the existing development process and keeping track of a variety of productivity data for comparison.

After completing the study, we conducted a usability workshop with a group of Nortel engineers to investigate the usability and effectiveness of the FDT used in the study further.

In the remainder of this section we explain the criteria we used to select a software system to be formalized and give an overview of this system.

2.2 System Selection

Choosing an appropriate system constitutes an important part of the study. As not all systems are worth engineering [42], 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. In our project, we looked for a system that was non-trivial but manageable, that is, had relatively stable requirements, was not too broad in scope, and was fairly self-contained, with loose coupling with the underlying system. The later requirement is necessary because of the scope of our project: we did not have the resources to adequately formalize the underlying system. Finally, since FDTs are most mature in the area of reactive systems, we wanted to measure the economic impact of the formalization on a reactive[2] system.

In addition, we wanted to work on a system that is representative of typical projects of the TorMAC division of Nortel. A group of Nortel engineers, consisting of software designers and software testers, decided that we should work on a subsystem of the Operation, Administration and Maintenance (OAM) software of a multimedia-messaging system connected to a private branch exchange (PBX). The subsystem, called ServiceCreator in our paper, is a voice service creation environment that lets PBX administrators build custom telephony applications. Such applications, e.g. an automated technical-support call redirector, can be built through a graphical workplace without worrying about the underlying complexity of a phone switch. We provide a (modified, due to the non-disclosure agreement with Nortel) view of ServiceCreator in Figure 1.

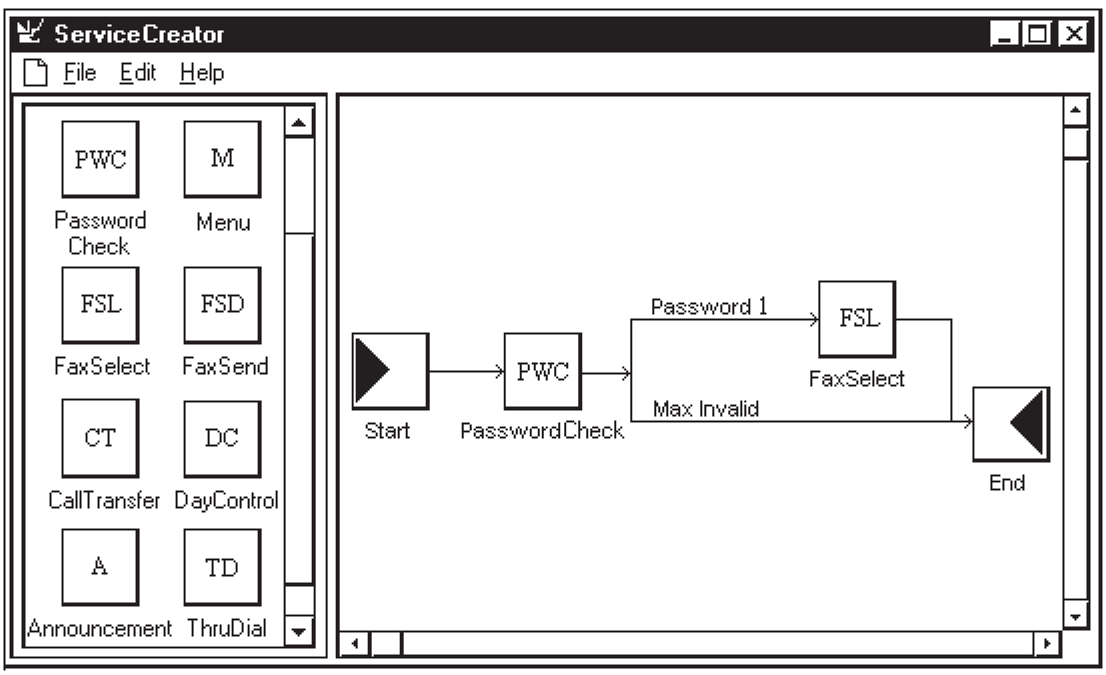


Fig. 1. Modified view of ServiceCreator.

The left panel of ServiceCreator shows some of the voice-service components available to administrators; the right panel shows the graphical workspace where

[2] Reactive systems are those where almost all inputs result in outputs that accurately indicate the state of the system [45].

administrators can connect predefined components, such as the voice menu, fax selection, call transfer and announcement, to build telephony applications. For example, a telephony application shown in the right panel lets callers select a password-protected fax document. 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 select the fax document associated with the *FaxSelect* component. She is directed to the *End* component if the maximum number of incorrect entries is reached. In both scenarios, the call session ends when the *End* component is reached. Lines connecting various components represent the potential control flow of the call session, and the actions performed by the caller in an active component determine the output path taken. For instance, in the component *PasswordCheck*, 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.

During run-time, the telephony application keeps track of a variety of persistent data about the call session such as the number of invalid password attempts and whether touch-tone keys have been received. Some components make use of this information for determining the appropriate output path. For instance, if a caller stays idle in a voice menu, the timeout routine for rotary callers is activated if no touch-tone keys have been received in the call session. In the other case, a timeout prompt is played to encourage the caller to take action. Thus, although many voice-service components are shown as distinct entities in the graphical workspace, they affect the behavior of each other by accessing and modifying persistent data during the actual execution.

ServiceCreator requires a "medium" level of assurance: while this software should be of higher quality than a typical office productivity application such as a word processor or a calendar, it is neither a safety-critical nor a high-availability system. The Nortel engineers made it clear that they do not attempt, or consider it necessary, to attain "zero defects" quality in systems like ServiceCreator. Instead, their goal is to ensure that the software meets customer expectations using the least amount of resources.

In our study, we analyzed the run-time behavior of 16 ServiceCreator components, described by an 80-page natural language specification (out of a total of 23 components with 97 pages of requirements); a brief description of these components is given in Table 1. In this paper we focus our discussion on the analysis of PasswordCheck, which has the low-to-medium complexity relative to the other components. We present a brief summary of this component here; the original Nortel specification is given in Appendix A.

The PasswordCheck component is described by a 5-page natural language specification. PasswordCheck, like any other component, has two perspectives: the Administrator perspective (*GUI*) allows the administrator to configure the component through a graphical interface, whereas the Caller perspective (*run-time*) represents the behavior of the component during a call. Throughout our study, we modeled only the run-time behavior of ServiceCreator components. Figure 2 shows the workspace view of this component. When a caller first enters

Component Name	Brief Description
Start	denotes the beginning of the application and performs the necessary initializations
End	terminates the application and performs the necessary cleanup
RotaryDial	directs rotary callers to the predefined attendant for assistance
Announcement	plays a voice prompt
SetLanguage	changes the default spoken language of the application
Menu	lets the caller make a touch-tone selection
PasswordCheck	uses predefined passwords to control access to certain parts of the application
DateControl	uses the current date to control access to certain parts of the application
DayControl	uses the current day of the week and predefined holiday schedules to control access to certain parts of the application
TimeControl	uses the current time to control access to certain parts of the application
FaxSelect	lets the caller select a fax document
FaxSend	sends the selected fax documents to another caller
CallTransfer	transfers the caller to a predefined phone extension
ThruDial	lets the caller transfer herself to another phone extension
ImportApp	represents the current application as a single voice-service component that can be embedded into another application
Continue	transfers the control flow from a terminated embedded application back to its parent application

Table 1. The 16 ServiceCreator components analyzed in our study.

PasswordCheck, an initial prompt is played. The digits entered by the caller are then validated 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 leaves the component using the *Max. Invalid* path 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* path 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 digit prior to entering PasswordCheck, she is assumed to be using a rotary phone and is transferred to the *RotaryDial* voice-service component (not shown in the Figure). 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.

Administrators can configure the behavior of the PasswordCheck component using a graphical interface. In particular, they can choose the initial prompt, define up to five numerical passwords for each component (at least one has to be defined), and set the *Number of invalid response tries* which controls the

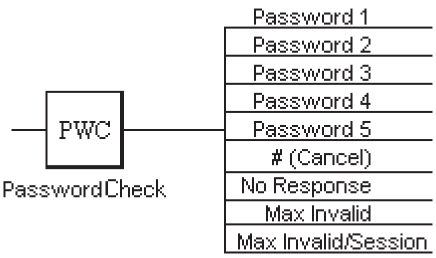


Fig. 2. The workspace view of PasswordCheck.

maximum number of invalid password attempts before the caller exists via the *Max. Invalid* output.

3 SDL

A successful formalization of a system in a commercial setting depends crucially on a modeling language, supported by an appropriate tool. The basic requirement we used for choosing a modeling language for a commercial setting was a simple notation that allows to create easily readable and reviewable artifacts. Moreover, since one of the overall objectives was to amortize the cost of creating a formal specification through testing, we looked for formal methods supported by tools enabling both modeling and testing. Then we used criteria put forward by Nortel engineers to find the most effective method supported by a well-developed tool. Some of the above criteria referred to methods ("readability", "scalability", "gentle learning curve"), and some to their supporting tools ("multi-user support", "version control"). For more information about this study, please refer to [51, 52].

The study chose the Specification and Description Language (SDL), supported by Telelogic SDT [46]. SDT is an integrated software modeling and testing tool suite that utilizes SDL for behavioral modeling and Message Sequence Charts (MSCs) [30] 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. Several SDL tools are available on the market: SDT, ObjectGeode, Cinderella, just to name a few. When choosing the appropriate tool, we were less concerned with the choice of a particular SDL tool, and more with the general modeling/testing paradigm. In this respect, most SDL tools were equivalent for our purposes.

SDL is a standardized language [31], based on extended finite state machines (EFSMs) and suitable for specifying reactive systems. In addition to its formal syntax and semantics, SDL also contains a recommended methodology guide [9]. Models can be created in a textual form or using a flowchart-like graphical notation; only the latter has been used in our study.

SDL models have three major layers: system, block and process. Each SDL *system* contains interconnected blocks; each *block* may contain multiple processes. Figure 3 shows the skeleton of an SDL model. The system and blocks layers are intended for organizing systems hierarchically. They allow to separate between the logic and the communications between different components within a large system. The process layer is used for defining the dynamic behavior of the system. The logic of the system is encoded into processes, which can be constructed using the SDL diagramming notation (see Figure 4).

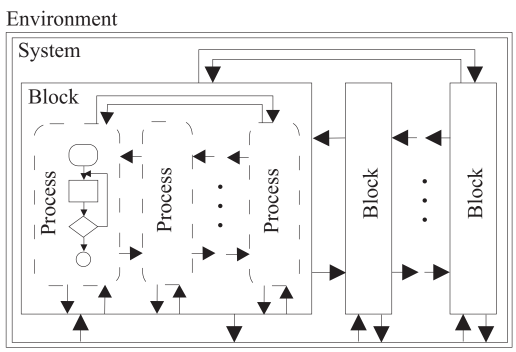


Fig. 3. SDL layers and channels.

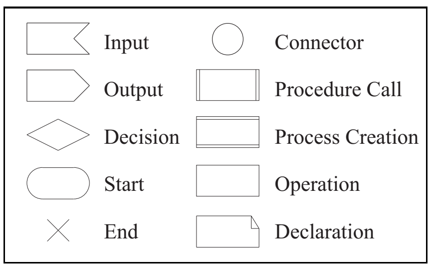


Fig. 4. Commonly used SDL symbols.

SDL *processes* are EFSMs that execute concurrently with equal priority. They communicate with each other and the environment, i.e. systems outside the SDL model, asynchronously by passing *signals* through *channels*; only point-to-point communication, i.e. one sender and one receiver, is supported. Because of the hierarchical structure of the models, channels may need to pass through several layers before reaching their final destination, which can either be the environment or other processes. The arrows in Figure 3 show the possible placement of the channels. Local variables and procedures can be used to describe the internal behavior of a process. Like any programming languages, SDL supports both simple and complex data types. More detailed information on SDL's syntax and semantics is available in [18, 39, 31, 52].

4 Modeling

In this section we describe the formalization of the ServiceCreator application in SDL. This formalization was undertaken by us in parallel with the actual development process. ServiceCreator was modeled as a 70-page SDL system in which the environment contained the underlying OAM software and messaging-system hardware. Out of the 16 voice-service components, 10 were modeled as separate SDL processes that communicated with the environment through a *main* SDL process which also incorporated the functionality of the five remaining components (e.g., *start* and *end*).

Figure 5 gives the high-level organization of the SDL model. It has a component *messagingSystem* that contains a block *SCApplication*, representing a ServiceCreator telephony application. Additional components of the messaging system, such as a user name database and a call-processing subsystem, can be modeled as SDL blocks and added to this SDL system. The block *SCApplication* contains the *main* process which administers control-flow of the telephony application, i.e. activates the appropriate voice-service component when a caller exits via an output path. The procedure *signalRedirect* routes signals between the environment and the active component.

The voice-service components are modeled as SDL processes that reside in an SDL package called *Common*. Inside an SDL package, a process is called a process type. This package construct allows the reuse of SDL processes by different SDL models. Thus, we can build models for various ServiceCreator applications without any modification to the SDL processes of the voice-service components. Such separation of concerns between the components and the applications of ServiceCreator also eases the task of maintaining the specification. There is a total of 10 process types, one for each separately-modeled voice-service component, and one package called *Common*. The complete SDL system uses 23 signals; eight of them external (used for communicating with the environment) and the rest internal.

The rest of this section is organized as follows: we discuss the formalization of one of the components, PasswordCheck, and the *main* process in Sections 4.1 and 4.2, respectively. For brevity, we do not include models of other components here; their complexity is similar to the described models. Section 4.3 discusses the methodology of modeling at the appropriate level of abstraction, whereas Section 4.4 lists the errors that were discovered during the formalization process.

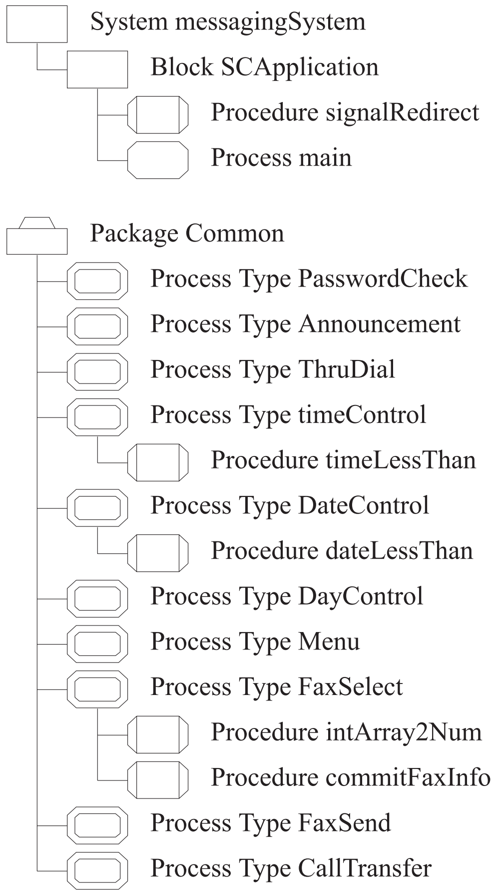


Fig. 5. A high-level overview of the SDL model of ServiceCreator.

4.1 SDL Model of PasswordCheck Block

Figures 15-18 of Appendix B show the 4-page SDL specification that we created for the PasswordCheck component. The SDL process shown in this diagram describes how PasswordCheck responds to the touch-tone keys from callers. Parameters to the SDL process, shown in the top-left corner of this Figure, represent persistent data of the component. For example, *globalState* describes the state of the messaging system, *promptMethod* is the initial prompt, and *maxRetry* is the maximum number of invalid password attempts. *pw!* and *IsPW!Enable*, with *!* ranging between 1 and 5, represent the numerical values of the five passwords and whether these are enabled, respectively. The explanations for local variables used in the process can be found in the the declaration symbol near the top-right corner of the diagram in Figure 15.

The process begins at the *Start* Symbol and clears the key buffer. The buffer may be modified by the Announcement component when the caller presses a key during the playing of the announcement. The process sends the signal *voice_0* together with the content of the initial prompt, to the environment to denote the action of playing the initial prompt. After activating the timer *noRespT* for detecting a possible idle timeout, the process waits for the caller's input, chosen among *, # or a digit key. In the first case, the process deactivates the timer, clears the timeout count, and sends *voice_0* with a help message to the environment. It also changes *globalState* to indicate successful interaction with a touch-tone caller, reinitializes the password entry and modifies the value of *isDigitEntered* to reflect this action. The scenario is similar when a digit key is received; however, the received digit *number* is accumulated in the variable *numberRecv*, and the value of the variable *isDigitEntered* is modified to indicate that a partial password has been entered. When the key # is detected, this variable determines whether the process should attempt a password match or exit prematurely. In the cases where the caller enters a correct password or requests a premature exit, the process sends a parameterized internal signal *done* to indicate the cause of termination. When a timeout occurs, the flow of control is transferred to the connector *noRespT*. PasswordCheck determines whether the caller is using a touch-tone phone by inspecting *globalState*. For rotary callers, it exits by sending the signal *done* and transferring the control to the RotaryDial component (this is not shown in the Figure). Otherwise, it plays a context-sensitive prompt that depends on whether the caller has begun entering the password. The prompt for the second timeout is not context-sensitive. After three consecutive timeouts, PasswordCheck aborts by sending the signal *done*.

4.2 The *main* Process: Control-Flow of the Application

The *main* process is responsible for modeling the control-flow information of the telephone application. It also acts as a gateway between voice-service components and the caller, enabling the communication between them using the procedure *signalRedirect*. We decided to implement a router instead of the direct communication to simplify the input/output interface of the block *SCApplication*.

Consider a sample ServiceCreator application shown in Figure 6. This application uses announcements to determine which path was taken out of the *PasswordCheck* component. If the user enters the password correctly, the appropriate announcement is played and the application exits. Similar action results from the user pressing the # (*Cancel*) key, not responding for a certain length of time, or entering the password incorrectly too many times. Note that this application includes the *Start* and the *End* components.

The control-flow information of the *main* process for the application in Figure 6 is given in Figure 7. This and most of the other figures in this section were created using Telelogic SDT. Initially, the process *main* initializes variables storing persistent data and then asynchronously invokes the SDL process for *PasswordCheck*; the configuration of this component is passed as a set of parameters. Such information includes persistent data (*globalState*), initial greeting

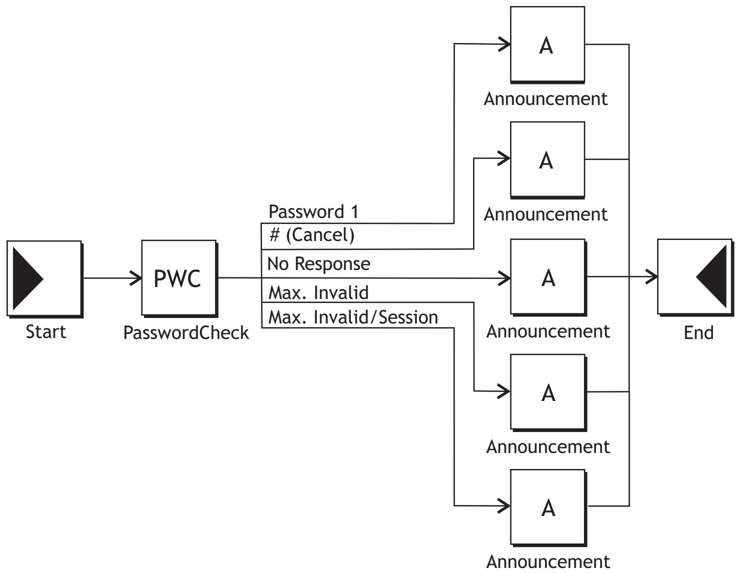


Fig. 6. The graphical view of a sample ServiceCreator application.

prompt (*prompt*), the maximum number of times a password can be reentered (*maxRetry*), whether each of the five passwords are enabled and what the values of the passwords are. In the actual implementation of ServiceCreator, the GUI dialog of Password validates these parameters. However, since we formalized only the run-time behavior for the components in this study, we needed to use the SDL process to ensure that these parameters were configured correctly; for example, we checked that at least one of the passwords has been enabled and that the value of this password has been identified. After invoking the PasswordCheck process, *main* executes the procedure *SignalRedirect* to redirect the communication between the callers and the ServiceCreator application. When the caller leaves the component, the procedure ends, and the variable *exitCode* indicates the output path taken. If the caller leaves PasswordCheck via *Cancel* path, *main* invokes the Announcement component (we removed invocation of this component on other PasswordCheck paths for brevity). These invocations are similar except that a different voice prompt (the value of *customPpt*) is passed to the Announcement component. Afterwards, *main* again executes *SignalRedirect*.

Note that the SDL process *main* hard-codes the parameters of the PasswordCheck and the Announcement components and the flow of control of the application. As a result, we needed different *main* processes to model ServiceCreator applications with different components or control-flow logic. SDT provides facilities for switching between the various *main* processes conveniently.

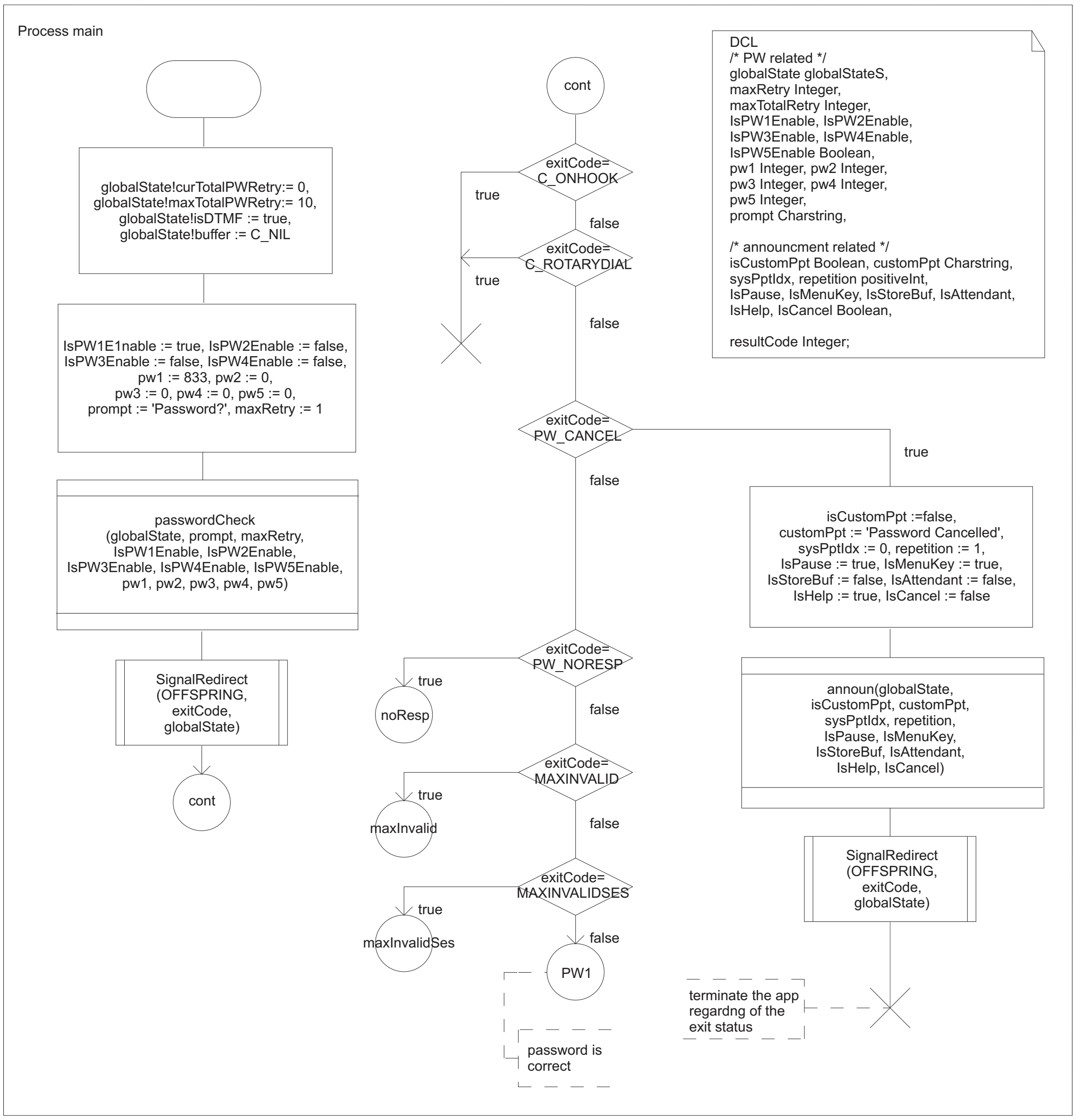


Fig. 7. The SDL *main* process of the application in Figure 6.

4.3 The Level of Abstraction

The modeling process was relatively straightforward as we encountered no major problems in modeling the voice-service of ServiceCreator; we also felt that a background in formal description techniques 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 point of view to reduce its complexity, while the exact behavior of the system needs to be modeled for deriving detailed 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 the details about parts of the behavior if the natural-language specification required it. For example, while modeling the PasswordCheck component, we represented the various timeouts by an SDL timer *timeout* (see Figures 15-18), as the actual length of the timeouts was relatively unimportant. We also did not include the actual content of the voice prompts in the model, or model the fact that PasswordCheck prompts are interruptible. We felt that this trivial behavior is already fully understood by the designers and the testers. Yet, to incorporate this requirement into our model, we would have needed an additional state every time PasswordCheck plays a voice prompt (see Figure 8 for an illustration), which would complicate the model significantly without adding much insight.

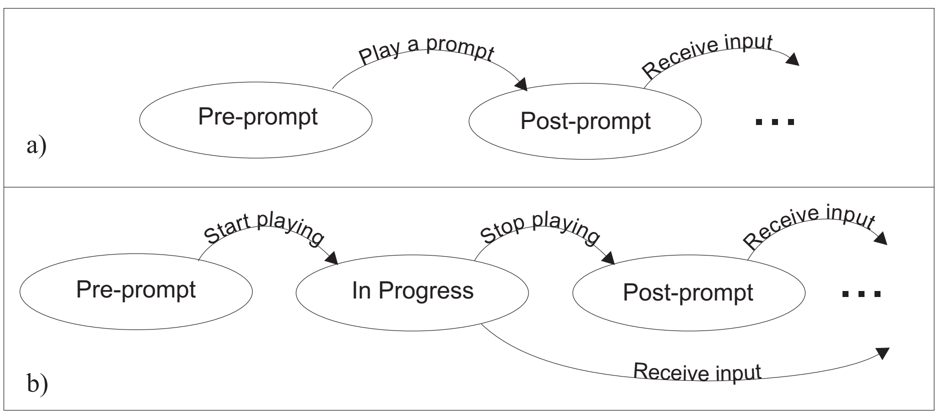


Fig. 8. Two possible ways to model the action of playing a voice prompt: a) a simplified approach used in the study, and b) a more accurate approach.

As mentioned earlier in this section, we only modeled the run-time behavior of the voice-service components, abstracting from other subsystems in the messaging system (e.g., the user name database); these were modeled as part of the environment. This abstraction simplified the SDL specification and allowed us to concentrate on modeling the core functionality of the system. The model also corresponded closer to the original specification. Clearly, the model did not reflect the complete system behavior; thus, if testers intended to use the test cases derived from this formal model for testing the implementation, they needed to fill in the missing information. We did, on the other side, structure our model so that the missing subsystems can be modeled as additional SDL blocks with virtually no changes to the rest of the specification.

Finally, we tried to simplify the specification further by modeling only those voice-service components that were complex or particularly important, leaving the conceptually simple components in their natural-language form. The latter included all GUI-related behavior of voice-service components. While these may also benefit from the formalization process, the leverage would be smaller, since designers can probably describe these requirements informally and still fully understand them. Not only did the omission of these components increase the efficiency of the entire formalization process, but the resulting model also became smaller.

4.4 Specification Problems Found by the Formalization

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 the Nortel engineers. In our study, we reported a specification error if

- a missing requirement affected the understanding of the system's behavior;
- a requirement conflicted with other parts of the specification; or
- a requirement was presented in an unclear or confusing manner.

The following are the problems that were found in the *original* specification of PasswordCheck; closely-related errors are grouped into one item.

GUI behavior

FG1 The numerical range of the number of invalid response retries, the range of the voice IDs, the range of the maximum number of invalid password entries, and the format of the voice-item name are not specified.

FG2 The types of initial greetings that are available to users to choose from are not mentioned.

FG3 The maximum and the minimum password lengths are not specified.

Note that we did not formally model the GUI-related behavior of Password-Check. The errors were identified by careful manual inspection, enforced by the need to model the run-time behavior.

Run-time behavior

FR1 Two different types of timeouts are mentioned in the specification, but none of them are specified numerically.

FR2 If an incorrect password is entered, it is unclear which prompt, if any, will be played. It is also unclear whether the initial greeting will be played again at this point.

FR3 Consider the case when a timeout occurs and a key (i.e. the help key) interrupts the timeout. If another timeout occurs again, it is unclear whether the first or the second delayed prompt is played.
FR4 It is unclear whether the attendant key (the *0* key) is enabled. If that is the case, then no password can start with a *0*.
FR5 It is never mentioned whether prompts are interruptible.
FR6 If two of the passwords in the password check block are the same (e.g. password 1 = password 2), it is not clear which exit path the password block will take when that particular password is entered, i.e. it is unclear whether some of the passwords have higher precedence than others.
FR7 If the maximum number of invalid retries or the maximum number of invalid retries per session are reached at the same time, it is unclear which error prompt is played. Furthermore, it is unclear if the PasswordCheck block is exited via the max. invalid or the max. invalid/session path in this case.

As the modeled components 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 Nortel engineers for clarifications. We estimated that these activities took as much time as the formalization process itself.

5 Testing

To obtain immediate benefits from the formalization process, we derived 120 MSCs from the 70-page SDL model for testing the implementation. The derivation was not automatic: as we "walked" through the SDL model manually, SDT recorded the interactions between the SDL system and its environment as MSCs. The Nortel team felt that the automation was not necessarily desirable since this exercise would give them confidence in the content and the coverage of the test cases.

5.1 Component-Based Testing

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 the full transition coverage of the corresponding SDL processes. Essentially, we created one ServiceCreator application model, such as the one shown in Figure 6, to test each component. In Figure 6, the PasswordCheck component is the test subject, and five Announcement components are used for identifying the output path taken by the caller.

Figure 9 shows a MSC that describes one particular interaction between a caller and PasswordCheck (refer to Figure 10 for a list of commonly used MSC symbols). The two participating objects start in *caller* and *pwChkInstance* states, respectively. The name *Process passwordCheck* above the object on the right associates this object with the SDL process in Figures 15-18.

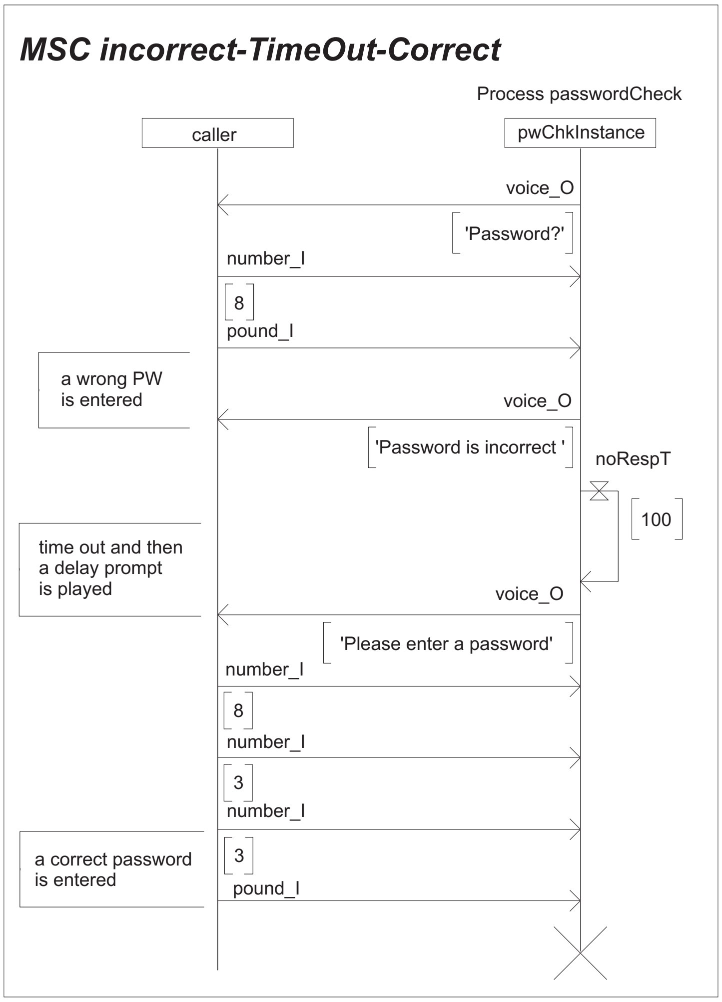


Fig. 9. A MSC created for the PasswordCheck.

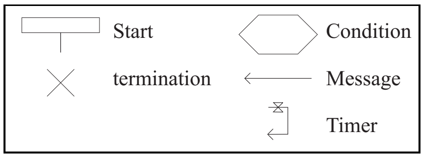


Fig. 10. Commonly used MSC symbols.

This MSC describes the case where a caller hears an initial prompt and then enters an incorrect password "8". After staying idle and hearing a delay prompt, she eventually enters a correct password "833". These interactions are represented by *Message* symbols with names that match the signals in Figures 15-18. Some of the interactions, such as the message *digit*, are parameterized with the actual parameters shown in square brackets. If the caller stays idle for 100 time units (indicated using a *timer* symbol) after the the "Password is incorrect" prompt, the system timeouts and resets its prompt to "Please enter a password".

A total of 17 component-based testcases were created for PasswordCheck. These are listed below:

T1 Select the system default initial prompt in the PasswordCheck dialog. After hearing the initial prompt, press the # key.

T2 Select a custom initial prompt. After hearing the initial prompt, stay idle until the timeout prompt is played.

T3 In the PasswordCheck dialog, enter a password but do not enable it. During run-time, key in that password to verify that it is properly disabled. Repeat this exercise for all five password entries.

T4 Enable a password and enter it in the PasswordCheck GUI dialog. During run-time , key in that password and press the # key. Repeat this exercise for all five password entries.

T5 Choose 'no prompt' in the PasswordCheck dialog. Enter two identical passwords in the PasswordCheck dialog and enable them. During run-time, key in that password and press the # key.

T6 Set the number of invalid password retries to two. Key in an incorrect password and press #. Then key in a correct password.

T7 Key in a correct password and wait for a timeout. The entered password should not be stored. Key in an incorrect password and wait for three timeouts.

T8 Set the maximum number of invalid password retries per session to five and the number of invalid response retires to one. Key in an incorrect password and wait for a timeout.

T9 Set the maximum number of invalid password retries per session to five and the number of invalid response retries to three. Key in an incorrect password and then press the # key. Repeat this three times.

T10 Set the maximum number of invalid password retries per session to two and the number of invalid response retires to three. Key in two incorrect passwords.

T11 Set the maximum number invalid password retries per session and the number of invalid response retires to one. Key in an incorrect password.

T12 Wait for a timeout. Then key in the first few digits of a correct password and then press the * key. The entered digits should not be stored. Key in the entire password and then press the # key.

T13 Try to set a password of length zero in the PasswordCheck dialog.

T14 Try to set an abnormally long password in the PasswordCheck dialog.

T15 During run-time, key in a large number of digits to test the maximum password length.

T16 Stay idle and wait for the rotary timeout.

T17 Press the *** key before keying in any digits. After keying in a few random digits, press the *** key. Wait for three timeouts.

5.2 Integration Testing

Some functionality of the system could be covered only by testing multiple components together, i.e. by the integration testing. 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 the caller was required to press a key *1* in the Menu component prior to entering the PasswordCheck component (see Figure 11). The MSC derived from this application is shown in Figure 12.

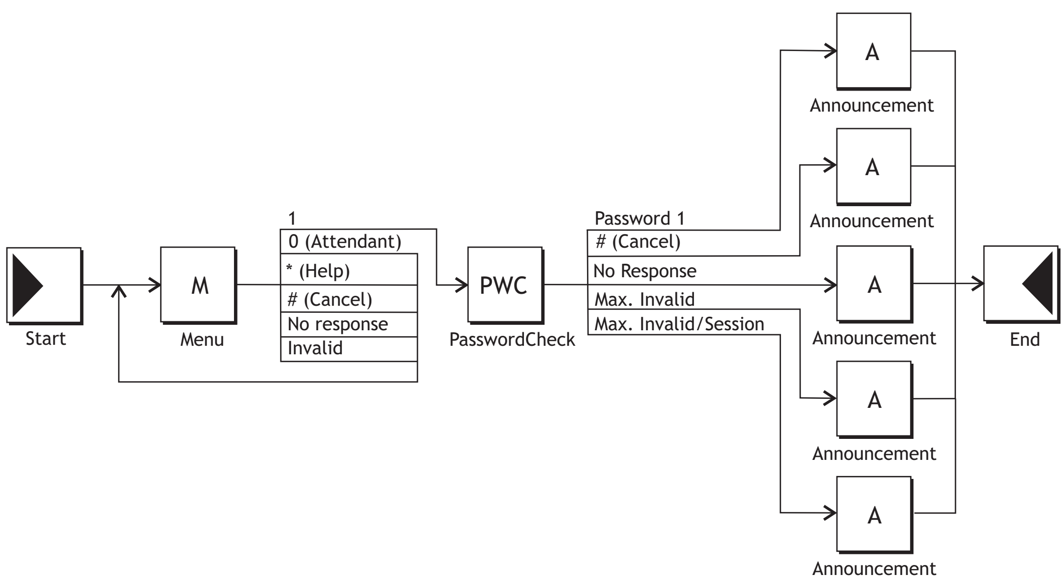


Fig. 11. The graphical view of the application created for testcase IT1.

The integration test cases for testing the PasswordCheck component are:

IT1 Create an application where the PasswordCheck component is not the first component of the application. Key in some inputs prior to entering the PasswordCheck component. Stay idle and wait for three timeouts.

IT2 To test key buffering, place the Announcement component (with key buffering enabled) before the PasswordCheck component. Set a password of at least two digits in the PasswordCheck GUI dialog. During run-time, when the announcement prompt is being played, interrupt it by keying in the first digit

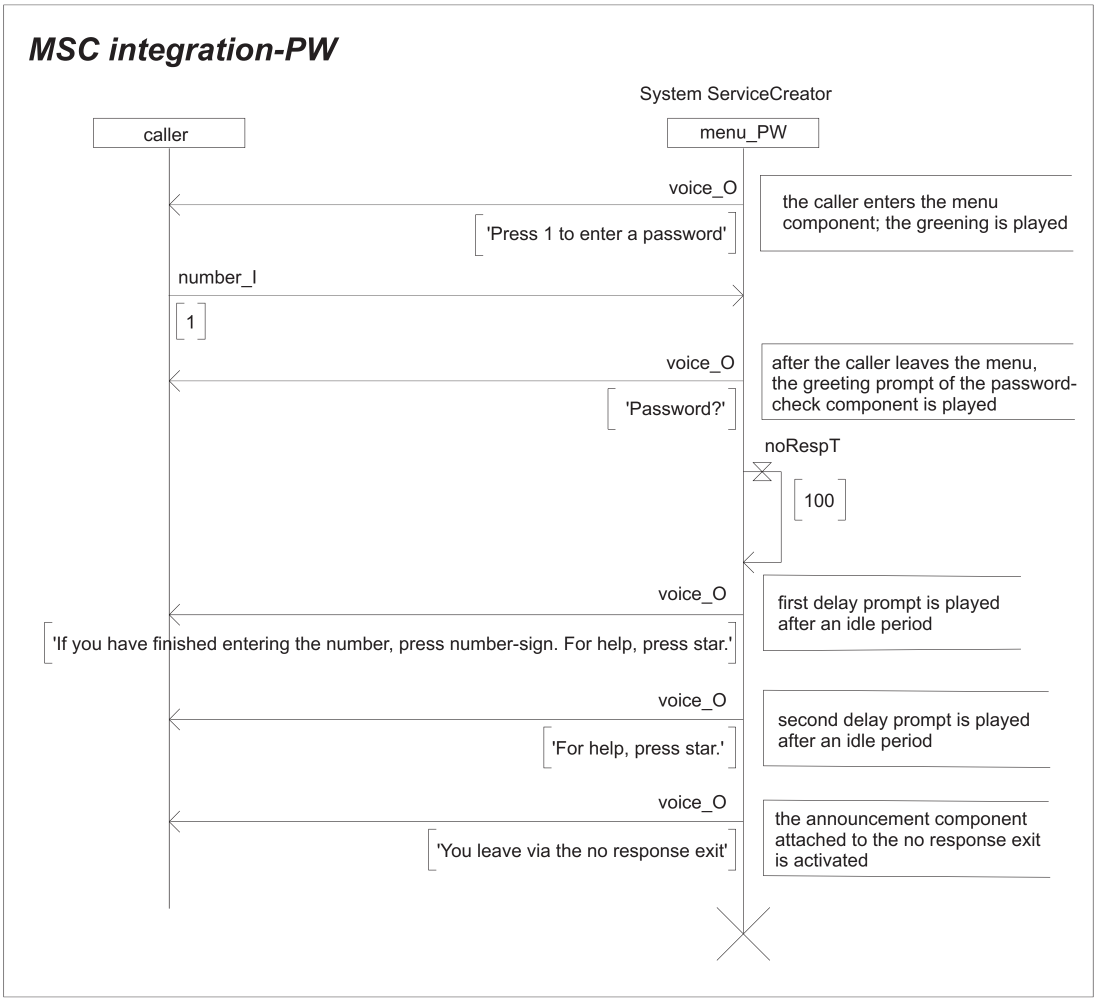


Fig. 12. An MSC showing interactions between a caller and a telephony application for testcase IT1.

of the password. Then, when the system prompts for a password, key in the rest of the password.

Such test cases can be derived using the following procedure:

- During the modeling phase, note the cases where the input comes not only from the environment but also from other components. If some input to component A comes from component B, we say that there is a relationship between A and B.

- After the modeling phase, use the resulting model to create test cases that specifically ensure that the relationship between the components is correctly implemented.

More than 20 integration test cases have been identified in the SDL model of ServiceCreator. Derivation of test cases for integration testing was the most labor-intensive part of this phase; it also required a fair bit of skill. We needed to identify relationships between the voice-service components and ensure that all such relationships were considered by the integration testing. Very often, the behavior involving accessing or manipulating the persistent data had to be tested using this type of test cases.

In addition, we set up integration test cases to verify if errors found during the formalization had propagated into the implementation. In particular, we needed these to verify errors FR6 and FR7 (see Section 4).

Using test cases derived from the formal model, we tested the first version of the implementation that was available to testers for official testing. We also kept track of the behaviors that did not agree with the specification, However, we intentionally did not report the errors that resulted from missing or unclear specifications but were rectified in the implementation (e.g., errors FR2, FR6 and FR7), as designers had already realized the existence of these problems. Clearly, the testers needed to interact with the developers to ensure that the problems have been fixed, spending extra time in the test case execution phase. The errors we found in the first version of the implementation are listed below; closely-related errors are grouped into a single item.

GUI behavior

TG1 Since the range of the voice ID is never specified, entering an ID that is outside the 16-bit-integer range (i.e. between -32768 and 32767) crashes the system.

TG2 If there is some information in the clipboard, pressing Ctrl-V (pasting information from the clipboard) in the password field fills it with an unknown password of length 43 which the system will accept.

TG3 The implementation allows the scenario in which no passwords are enabled, which conflicts with the specification.

TG4 If only the fifth password of the PasswordCheck block is specified, the system incorrectly interprets that no passwords are specified.

TG5 Entering a password with more than 12 characters triggers a GUI system crash.

As we left the project, all of these problems have still not been fixed.

Run-time behavior

TR1 The custom prompt cannot be played. Moreover, the content of the delay prompts is different from what is given in the specification.

TR2 There is only one delay prompt (hence, two timeouts before the disconnect) while the specification specifies that there should be two delay prompts.
TR3 If no key is entered, the delay prompt that follows a help prompt is missing.
TR4 The rotary timeout does not occur.
TR5 The system misses digits that are entered when the initial prompt is being played.
TR6 The system does not behave according to the limit on the number of invalid response retries set by the administrator.
TR7 Entering a password that is longer than 43 digits triggers a server crash. The PasswordCheck block accepts a password that starts with a zero; however, during run-time, the system can never verify such a password.
TR8 Typically, the key entered during an announcement is buffered and is used as input for the next component block. However, the PasswordCheck block ignores the input and clears the buffer. (This is actually the intended behavior but it is never mentioned in the requirements document.)

The last error has been found via integration testing. All errors except TR4, TR5 and TR6 were not fixed by the time we finished this project.

6 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 also crucial to identify the qualitative factors (e.g., perceived usability of SDL and commitments from the development team) and the limitations of the study in order to reach accurate and unbiased conclusions.

Unfortunately, Nortel engineers did not keep track of many essential metrics and, due to the exploratory nature of our study, did not allow us to create a controlled environment where such metrics could be obtained. In particular, we could not determine the exact amount of time it took to fix a bug, if it was found during the inspection vs. design vs. testing phase. The lack of metrics significantly impaired our ability to draw quantitative conclusions.

6.1 Changes to Development Process

When we began our formalization process, the development of ServiceCreator was in the testing phase: the requirements specification went through five manual reviews, the design and implementation were completed, and the Nortel testers were testing the system as the designers were fixing the reported errors. We could not investigate the effect of formalization on the entire development process; instead, starting with the inspected requirements specification, we modeled the behavior of several components of ServiceCreator in parallel with the existing development process (see Figure 13). This setting was compatible with the expected use of SDL models where only certain requirements are selectively modeled to complement the natural-language specification. In addition, the modeling process was similar to performing a rigorous requirements inspection.

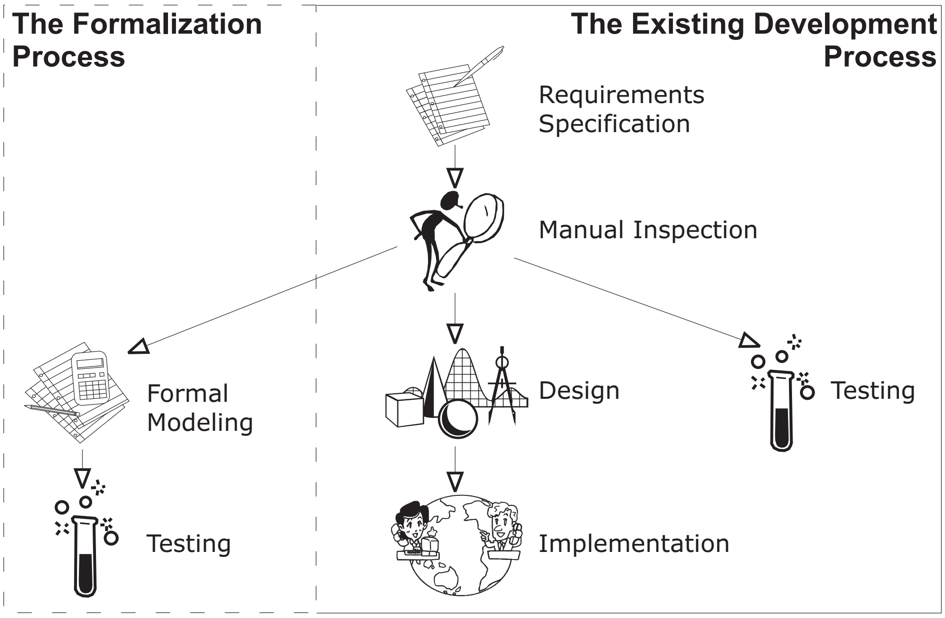


Fig. 13. The formalization process and the existing development process.

6.2 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). Effort measurements in this table are approximated to the nearest person-day.

Productivity Data	Study	Existing Process
Time to model (person-days)	11	N/A
Time to inspect (person-days)	N/A	50
Number of specification errors reported	56	N/A
Number of test units	269	96
Time to create test units (person-days)	7	7
Time to execute test units (person-days)	5	14
Number of implementation errors identified	50	23

Table 2. Productivity Comparisons.

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. While the modeling task did not have an equivalent in the actual development process, manual inspection was a similar activity that was also performed at the completion of the specification phase. Certainly, the formalization process was not meant to be a complete replacement. However, if a large number of specification errors were identified in a relatively short amount of time, the modeling task could be considered as a way to decrease the time for inspection. (We discuss this point later in this section.)
- 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) in roughly the same amount of time, 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 was that the test units from Nortel were sometimes vaguely specified (see Figure 14 for an example); the missing details contributed to a decrease in the number of test units reported.

Call the application and simulate a number of hacker attempts (such as common passwords and misspelled passwords). Verify that the application terminates the call session after the maximum number of password attempts is reached.

Fig. 14. A Nortel test case for testing the password-check component.

- The time needed for test unit execution in our study was much smaller for two major reasons: the derived test units were more detailed and thus easier to execute, and it was observed that Nortel engineers spent a lot of time revising the existing test cases because of the changes in requirements and creating more detailed test scenarios based on the vaguely specified test units. However, due to tight schedules, most testers did not document these extra test units until the end of the entire testing phase, which spanned over more than four months. They admitted that some of these test units would inevitably be lost, contributing to a decrease in their total number.
- The number of implementation errors identified in the study was two times larger than that of the existing development process. Many of them were missed because testers created test cases from an incorrect and incomplete specification, as indicated in the third row of Table 2. Problems such as incompleteness propagate to test cases and affect the test coverage. In fact, 18 of the 50 problems could be linked to problems in the specification.

Their criticality ranged from the minor ones, dealing with the availability and the interruptability of voice prompts, to the critical ones, affecting the core functionality of the voice-service components or causing the system to crush. Most of these errors resulted from undocumented assumptions or incorrect/missing error handling.

To obtain an accurate cost/benefit figure, we needed to collect additional statistics such as an average cost to fix a requirements error discovered during the inspection and the implementation phases. As we mentioned above, the Nortel development team did not keep track of such metric; however, a conservative cost/benefit estimation is still possible. Without taking the improvement in software quality into account, we can estimate the cost of formalization by subtracting the time of the modeling task from the direct savings in

- the test unit creation (0 person-days),
- the test unit execution (9 person-days), and
- the manual inspection.

The formalization process did not include a manual inspection phase, whereas the actual development took 50 person-days for it. The modeling task would come at no cost if it were to reduce the manual inspection by 2 person-days, or 4%. Of course, the actual cost/benefit figure is significantly more promising if the long-term benefits, coming from a better quality of the product, ease of maintenance and regression testing, and an ability to reuse a good specification, are taken into account.

6.3 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 the test engineers is that the task of creating test cases is complicated by incomplete or missing information in the specifications. 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 during the design and the implementation phases. This is an area where SDL can potentially significantly improve the development process.
- As with any other formal specification technique, a successful integration of SDL into the development process requires a firm commitment 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 justify the extra effort.

- Compared to other formal modeling techniques, the strengths of SDL lie in its ease of use and the ability to express nontrivial behavior in a reviewable notation. Unlike many other formal modeling languages, SDL does not require an explicit use of formal logic. The graphical user-friendly notation allows developers without a strong mathematical background to effectively create EFSMs. Compared to natural language specifications, such EFSMs give a much better sense of completeness, allowing to easily detect missing scenarios (e.g., causing errors FR4, FR5, FR6 and FR7 in Section 4.4) and ambiguities (e.g. causing errors FR1 and FR2, FR3). However, SDL tends to blur the line between requirements and designs. If proper abstractions are not applied, the model may become too detailed and unnecessarily large, possibly duplicating the design effort.

6.4 Lessons Learned

As expected, we observed an increase in software quality and productivity, in terms of the number of test cases generated, the time for creating and executing them, and the number of errors found in the requirements and implementation. Moreover, it was possible to obtain these benefits without lengthening and possibly while shortening the development cycle. The success of the project was in

- finding a suitable system — the impact of formalization differs significantly between various types of systems. It was important for us to choose a system that had a good potential to benefit from the formalization and yet was relevant to and representative of typical Nortel systems.
- using a well-chosen formal description method — it was important that we chose a method that would tackle the problems faced by the Nortel development team and was able to formalize the requirements of the chosen system effectively, The method also needed to fit into the Nortel environment because the Nortel engineers would be the eventual users.
- taking a lightweight formalization approach — we decreased the cost of formalization by employing a notation that was easy to use and review, formalizing only selected components, and staying at a fairly high level of abstraction. ServiceCreator, like the majority of commercial systems, did not require formal verification. Because of that, we felt that it was unnecessary and even counter-productive to specify every detail of the system. In fact, our use of SDL showed that if the verification is not required, it is not essential to use a modeling language with a fully-defined formal semantics to achieve immediate and measurable benefits.

- leveraging the cost of formalization on the testing phase — FDTs inevitably increase the length of the requirements phase. By minimizing the cost of formalization and amortizing the cost of creating the model over the time-intensive software testing phase, the formalization process came virtually at no cost. The savings came from shortening the time for creating and executing test cases.
- the formalization was done as a complement rather than a replacement of the existing natural language specification. No major changes to the existing development process were required; in fact, this approach allowed the incremental use of the formalization, that is, the ability to stop at any time and get partial benefits from partial modeling. Certainly, successful integration of SDL, or any other FDT, into the development process would require a full support of the entire development team and the ability of the engineers to apply the method effectively.

6.5 Limitations

The project had a number of shortcomings:

- It would be difficult [53] to generalize our findings outside the Nortel domain, since they would depend on the current development methodology, types of applications and the choice of a modeling language/tool.
- We did not integrate SDL into the actual development process to observe the impact of the formalization process on the entire development cycle. Nortel engineers were not immediately involved in the modeling of ServiceCreator.
- The fact that we had a background in FDTs and were never under any development pressure might have positively affected the quality of the formal model and thus the derived test cases. In addition, our lack of prior knowledge in the telecommunications domain might have helped us detect subtle requirements errors [5].
- We did not fully quantify our findings because the Nortel personnel did not allow us to create a controlled environment where we could collect some essential metrics.

7 Conclusion and Future Work

In this case study we formalized the behavior of a multimedia-messaging system in a commercial setting. The success of the study was in finding a representative system, carefully selecting a suitable modeling method, and taking a lightweight formalization approach. Although we did not have access to some development metrics to fully quantify our findings, the results of the study clearly show that software requirements can be formalized effectively and economically, yielding significant improvements over the existing development process.

The majority of formalization studies concentrate only on increasing software quality. Among the few exceptions that we found, Brookes [8], Pfleeger [41] and Garbett [21] focus on the quantitative benefits of applying FDTs instead of traditional development techniques to safety-critical systems. In the commercial setting, a number of researchers have successfully applied formalization techniques, e.g. [25, 27, 36, 23, 1]. However, these studies did not collect compelling quantitative information. A notable exception is the study by Bicarregui [6]. It uses FDTs in developing a part of a commercial application integration product, "middleware". This study shows that FDTs can help detect requirements errors earlier, thus indirectly implying that long-term cost savings should result.

Our study is not unique in applying SDL to the telecommunications domain. SDL has been used from the traditional use of protocol specifications [4, 44, 19] to high level specification [35], prototyping [49], design [38], code generation [20], and testing [24] of telecommunications applications. The goal of these studies was different: they were aimed at investigating technical advantages or a feasibility of SDL in a given environment, or were emphasizing only one of the development activities.

Finally, while writing this paper, we learned about a case study done as part of the ESSI/SPACES project [14]. This study was set up to measure the improvement of development processes through automatic code and test generation from SDL models using ObjectGeode. There, SDL was applied to the development of space shuttle software, with results similar to ours.

For the future work, we plan to address some of the limitations of the project and expand the scope to focus on concerns that did not receive full attention in this project. By conducting more relevant studies that show both the benefits and the shortcomings of FDTs objectively, we hope to encourage more practitioners to experiment with FDTs. Our ultimate goal is to conduct control experiments demonstrating the economic effectiveness of FDTs.

- **More studies.** We exemplified the benefits of FDTs by comparing the productivity data of our project with that of the existing development process. Nortel and the University of Toronto plan to set up another joint project where SDL will be applied by the Nortel engineers. We will observe the process and provide help, if necessary. This will let us further access the usability of SDL and examine the effect of formalizing requirements on the development cycle. In fact, it is essential to have two control groups, so that the productivity comparisons can be made. If we have another Nortel team work in parallel, using the existing development technique, the comparisons can be much more direct and accurate.
- **Data collection.** The exploratory nature of our study prevented us from having an adequate data collection process. In conducting future studies, we must ensure that a proper process to collect productivity data is in place. Exact measurements of the number of requirements errors discovered at various stages of the development and the cost to fix a requirements error in the implementation are essential for quantifying the benefits of FDTs.
- **Replication in other domains.** Since our current results are based on a single study, we need to conduct more case studies to investigate the use of SDL in telecommunications systems of different complexity to validate the

results. To learn more about limitations and feasibility of applying FDTs in a commercial setting in general, we should also experiment with applying FDTs to other application domains using the basic principles that proved to be effective in our project, e.g., derive test cases or generate code from formal models and use FDTs in a lightweight manner. Unfortunately, the success of our study depended a lot on the effectiveness and maturity of tools [12], and we may not get similar results in another domain that does not have equally good tools.

- **Verification.** Given the favorable results from our project, another interesting direction is to explore the use of rigorous techniques in verifying selected properties of complex subsystems. Verification can encompass system-independent properties, such as freedom of deadlock or safety and security policies. Informal conversations with the engineers indicate that such techniques will be very helpful if they are applied in a cost-effective manner.

Acknowledgments

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, and Steve Easterbrook for his careful reading of previous drafts of this paper. This work was supported by Nortel Networks, NSERC and Ontario Graduate Scholarship (OGS).

References

1. Thomas Arts and Mads Dam. "Verifying a Distributed Database Lookup Manager Written in Erlang". In J. Wing and J. Woodcock, editors, *Proceedings of World Congress on Formal Methods in the Development of Computing Systems (FM'99), LNCS 1708*, pages 682–700, September 1999.
2. K. Beckman, N. Coulter, S. Khajenoori, and N. Mean. "Collaborations: Closing the Industry-Academia Gap". *IEEE Software*, 14(6), November 1997.
3. Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. "Meteor: A Successful Application of B in a Large Project". In J. Wing and J. Woodcock, editors, *Proceedings of World Congress on Formal Methods in the Development of Computing Systems (FM'99), LNCS 1708*, pages 369–387, September 1999.
4. F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification.* Prentice Hall, 1991.
5. D.M. Berry. "The Importance of Ignorance in Requirements Engineering". *Journal of Systems and Software*, 28(1):179–184, 1995.
6. Juan Bicarregui, Jeremy Dick, and Eoin Woods. "Quantitative Analysis of an Application of Formal Methods". In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 60–74. Springer-Verlag, March 1996.
7. Jonathan Bowen and Victoria Stavridou. "The Industrial Take-up of Formal Methods in Safety-Critical and Other Areas: A Perspective". In J.C.P. Woodcock and P.G. Larsen, editors, *Proceedings of FME'93: Industrial Strength Formal Methods, LNCS 670.* Springer-Verlag, 1993.

8. T.M. Brookes, J.S. Fitzgerald, and P.G. Larsen. "Formal and Informal Specifications of a Secure System Component: Final Results in a Comparative Study". In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 214–227. Springer-Verlag, March 1996.
9. CCITT, Geneva. *Appendix I to Recommendation Z.100—SDL Methodology Guidelines*, June 1993.
10. Michel Chaudron, Jan Tretmans, and Klaas Wijbrans. "Lessons from the Application of Formal Methods to the Design of a Storm Surge Barrier Control System". In J. Wing and J. Woodcock, editors, *Proceedings of World Congress on Formal Methods in the Development of Computing Systems (FM'99), LNCS 1709*, pages 1511–1526, September 1999.
11. M. Chechik. *Automatic Analysis of Consistency between Requirements and Designs*. PhD thesis, University of Maryland, College Park, Maryland, December 1996.
12. M. Chechik and A. Wong. "Formal Methods When Money Is Tight". In *Proceedings of the First Workshop on Economics-Driven Software Engineering Research (EDSER-1)*, Los Angeles, May 1999. (affiliated with ICSE-99).
13. E.M. Clarke and J. Wing. "Formal Methods: State of the Art and Future Directions". *ACM Computing Surveys*, 28(4):626–643, December 1996.
14. Eric Conquet and Jean-Luc Marty. "Formal Design for Automatic Coding and Testing: The ESSI/SPACES Project". In J. Wing and J. Woodcock, editors, *Proceedings of World Congress on Formal Methods in the Development of Computing Systems (FM'99), LNCS 1708*, pages 57–75, September 1999.
15. D. Craigen, S. Gerhart, and T. Ralston. "An International Survey of Industrial Applications of Formal Methods". *U.S. National Institute of Standards and Technology*, March 1993.
16. J. Crow and B.L. Di Vito. "Formalizing Space Shuttle Software Requirements". In *Workshop on Formal Methods in Software Practice*, San Diego, California, January 1996.
17. Steve Easterbrook, Robyn Lutz, Richard Covington, John Kelly, Yoko Ampo, and David Hamilton. "Experience Using Lightweight Formal Methods for Requirements Modeling". *IEEE Transactions on Software Engineering*, 24(1):4–14, January 1998.
18. Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *"SDL: Formal Object-Oriented Language for Communicating Systems"*. Prentice Hall, New York, 2 edition, April 1997.
19. Christian Facchi, Markus Haubner, and Ursula Hinkel. "The SDL Specification of the Sliding Window Protocol Revisited". Technical Report TUM-I9614, Technische Univerität München, 1996.
20. M.W. Froberg. "Automatic Code Generation from SDL to a Declarative Programming Language". In *Proceedings of the Sixth SDL Forum*, Darmstadt, Germany, October 1993.
21. P. Garbett, J.P. Parkes, M. Shackleton, and S. Anderson. "Secure Synthesis of Code: A Process Improvement Experiment". In J. Wing and J. Woodcock, editors, *Proceedings of World Congress on Formal Methods in the Development of Computing Systems (FM'99), LNCS 1709*, pages 1816–1835, September 1999.
22. M. Van Genuchten. "Why is Software Late? An Empirical Study of Reasons for Delay in Software Development". *IEEE Transactions on Software Engineering*, 17(6):582–590, June 1991.
23. Fred Gilham, R.A. Riemenschneider, and Victoria Stavridou. "Secure Interoperation of Secure Distributed Databases: An Architecture Verification Case Study".

In J. Wing and J. Woodcock, editors, *Proceedings of World Congress on Formal Methods in the Development of Computing Systems (FM'99), LNCS 1708*, pages 701–717, September 1999.

24. M. Grasmanis and I. Medvedis. "Approach to Behaviour Specification and Automated Test Generation for Telephone Exchange Systems". In *Proceedings of the Fifth SDL Forum*, Glasgow, Scotland, September 1991.
25. Anthony Hall. "Using Formal Methods to Develop an ATC Information System". *IEEE Software*, 13(2), March 1996.
26. 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.
27. Jonathan P. Hoare. "Application of the B-Method to CICS". In M. G. Hinchey and J. P. Bowen, editors, *Applications of Formal Methods*, pages 97–124. Prentice Hall International Series in Computer Science, 1995.
28. C. Michael Holloway and Ricky W. Butler. "An Invitation to Formal Methods: Impediments to Industrial Use of Formal Methods". *IEEE Computer*, 29(4):25, April 1996.
29. moderator Hossein Saiedien. "An Invitation to Formal Methods". *IEEE Computer*, pages 16–30, April 1996.
30. ITU-T. "ITU-T Recommendation Z.120: Message Sequence Chart (MSC)". *ITU-T*, 1993.
31. ITU-T. *Specification and Description Language (SDL): ITU-T Recommendation Z.100*. International Telecommunication Union, 1996.
32. Daniel Jackson and Jeannette Wing. "Lightweight Formal Methods". *IEEE Computer*, April 1996.
33. 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.
34. Steve King, Jonathan Hammond, Rod Chapman, and Andy Pryor. "The Value of Verification: Positive Experience of Industrial Proof". In J. Wing and J. Woodcock, editors, *Proceedings of World Congress on Formal Methods in the Development of Computing Systems (FM'99), LNCS 1709*, page 15271545, September 1999.
35. L. Mansson. "High Level Specification of a Telecom Application with SDL'92". In *Proceedings of the Sixth SDL Forum*, Darmstadt, Germany, October 1993.
36. Peter Mataga and Pamela Zave. "Multiparadigm Specification of an AT&T Switching System". In M. G. Hinchey and J. P. Bowen, editors, *Applications of Formal Methods*, pages 375–398. Prentice Hall International Series in Computer Science, 1995.
37. John McDermid, Andy Golloway, Simon Burton, John Clark, Ian Toyn, Nigel Tracey, and Sam Valentine. "Towards Industrially Applicable Formal Methods: Three Small Steps, and One Giant Leap". In *Proceedings of the Second International Conference on Formal Engineering Methods*, pages 76–88, December 1998.
38. M.Kooij and L. Provoost. "Industrial Report on the Use of Abstraction in SDL/MSC". In *First Workshop of the SDL Forum Society on SDL and MSC*, Berlin, Germany, June 1998. alcatel.
39. Anders Olsen. *"Systems Engineering Using SDL-92"*. Elsevier Science, New York, January 1994.
40. D.L. Parnas. "Some Theorems We Should Prove". In *Proceedings of 1993 International Conference on HOL Theorem Proving and Its Applications*, Vancouver, BC, August 1993.

41. Shari Lawrence Pfleeger and Les Hatton. "Investigating the Influence of Formal Methods". *IEEE Computer*, 30(2):33–43, February 1997.
42. Roger S. Pressman. "Can Internet-Based Applications be Engineered?". *IEEE Software*, 15(5), September 1998.
43. F. Randimbivololona, J. Souyris, P. Baudin, A. Pacalet, K. Raguideau, and D. Schoen. "Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach". In J. Wing and J. Woodcock, editors, *Proceedings of World Congress on Formal Methods in the Development of Computing Systems (FM'99), LNCS 1709*, pages 1798–1815, September 1999.
44. A. Sarma. "Modeling Broadband ISDN Protocols with SDL". In *Proceedings of the Fifth SDL Forum*, Glasgow, Scotland, September 1989.
45. Tony Savor and Rudolph E. Seviora. "Toward Automatic Detection of Software Failures". *IEEE Computer*, 31(8):68–74, August 1998.
46. Telelogic. "Telelogic SDT Home Page". `http://www.telelogic.com/solution/tools/sdt.asp`, September 1998.
47. Walter F. Tichy. "Should Computer Scientists Experiment More?". *IEEE Computer*, 31(5), May 1998.
48. Steve Tockey. "A Missing Link in Software Engineering". *IEEE Software*, 14(6), November 1997.
49. H.J. Vgel, W. Kellerer, S. Karg, M. Kober, A. Beckert, and G. Einfalt. "SDL based prototyping of ISDN-DECT-PBX switching software". In *First Workshop of the SDL Forum Society on SDL and MSC*, Berlin, Germany, June 1998.
50. Debora Weber-Wulff. "Selling Formal Methods to Industry". In J.C.P. Woodcock and P.G. Larsen, editors, *Proceedings of FME'93: Industrial Benefit of Formal Methods, First International Symposium of Formal Methods Europe*, pages 671–678, Odense, Denmark, April 1993. Springer-Verlag.
51. Andre Wong. "The Diary of the Formal-Method Survey". `http://www.cs.toronto.edu/~andre/progress.html`, September 1998.
52. Andre Wong. "Formalizing Requirements in a Commercial Setting: A Case Study". Master's thesis, University of Toronto, Department of Computer Science, Toronto, ON, Canada, May 1999.
53. Marvin V. Zelkowitz and Dolores R. Wallace. "Experimental Models for Validating Technology". *IEEE Computer*, 31(5), May 1998.

A Original Specification of PasswordCheck

We present the original specification of the PasswordCheck component of the ServiceCreator here.

> **Disclaimer:** The following text replicates the original Nortel specification for the PasswordCheck component. However, we removed references to Nortel products and modified several figures to comply with Nortel's non-disclosure agreement. We have retained the conceptual (e.g., inconsistencies and ambiguities) and stylistic (e.g., grammatical and spelling mistakes) errors present in the original specification.

The original description also included Figure 2, given in Section 2.

Functional Overview

The PasswordCheck block controls access to certain parts of an application by prompting callers to enter the password before being allowed to continue. Like all other blocks, the administrator places this block anywhere in an application. In defining the block, the administrator enters the password that callers must enter. The administrator is then responsible for remembering the password and distributing it to those people requiring it.

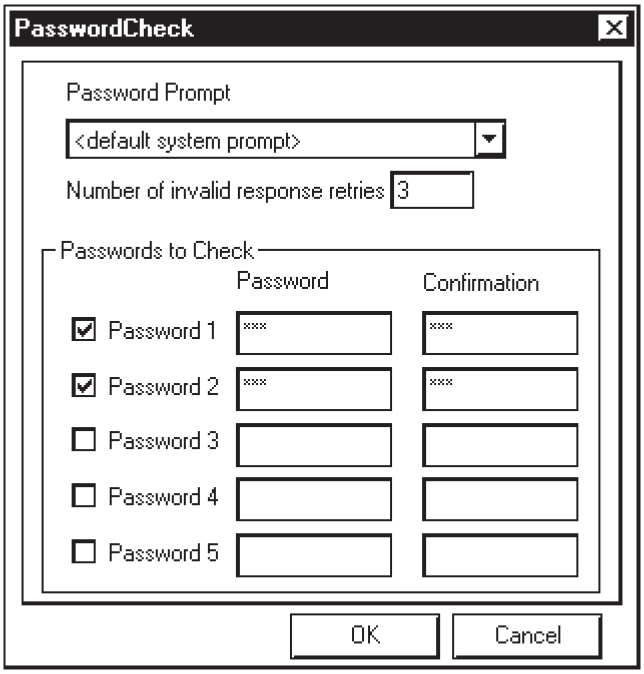
The PasswordCheck block has the following outputs:

- Valid Password (1 through 5)
  — this output indicates that the caller entered one of the valid passwords specified in the block's parameters, and exits on the respective output.
- Max Invalid
  — this output indicates that the caller entered the maximum number of invalid passwords
  — the caller should not be permitted to proceed
- Max Invalid Session
  — The caller has reached the maximum number of invalid password attempts allowed per session as set in the session profile.
- No response
  — the caller has made no response before the timeout.
- # (Cancel)
  — this output indicates that the caller cancelled the password entry by pressing the number-sign key (#) without pressing any digits
  — the caller should not be permitted to proceed

Administrator perspective

To setup the PasswordCheck block, the administrator must complete these steps:

1. The prompt played to the caller on entering the block must be defined and selected.
2. At least one of the application passwords must be selected. To select a password, the checkbox to the left of the desired password must be checked.
3. The password must be entered for all passwords that are enabled, and the respective outputs must be connected. To enter a password the administrator first types the password (only numbers 0-9 are allowed). The administrator must enter the password again in the confirmation text edit box to ensure that the password was not mistyped. The numbers in the password will be echoed to the display as the asterisk character (*) to ensure proper security.The password will be encrypted prior to storage on the disk.



Caller perspective

Immediate Prompt On arrival at the PasswordCheck block, the caller may hear a custom recorded prompt or the system default prompt. The greeting is intended to orient the caller and provide information on how to enter an extension number or name. If the system default prompt is chosen, the system plays: "Password:?"

Delayed Prompts The caller may experience one of two types of treatment if no key is pressed when the caller is in the PasswordCheck block. If the caller has arrived at the PasswordCheck block without yet pressing any keys in the call, the no response condition is treated as a strong indication that the caller is not calling from a touch-tone phone. In order to avoid annoying rotary-dial callers, the PasswordCheck block exits after the first Command Entry timeout period. The application jumps to the Rotary Dial block and follows its output to the next block. Thus the administrator decides the handling for these "Rotary Dial Timeout" cases. Typically the call would be transferred to an attendant.

In the second case, where the PasswordCheck block is reached after keys have already been pressed during the call (e.g. the PasswordCheck block is called from a Menu), the block follows a more extensive timeout sequence. After the first Command Entry timeout period passes, a delayed prompt is played (e.g. "Please enter the password followed by number sign"). A second Command Entry timeout is followed by a second delayed prompt (e.g. "For help, press star").

Finally, a Short Disconnect timeout period follows. When this time lapses, the PasswordCheck block exits on the No Response output without playing any prompt.

Delayed prompts are context sensitive. If the caller has begun to enter the password, the delayed prompts will urge the caller to continue:

"*If you have finished entering the number, press number-sign. For help press star.*"

If nothing has yet been entered, the caller is encouraged to star:

"*Please enter the password followed by number-sign.*"

Help

Pressing star in the PasswordCheck block plays the help prompt:

"*You have selected a service which requires a password. Please enter the password, followed by number-sign.*"

"*To cancel, just press number-sign.*"

This same prompt is played regardless of whether part of the password has already been entered or not. If part of the password has been entered when the star key is pressed, it has the effect of clearing the entry thus far.

Cancel If the caller presses the number-sign (#) key without entering any data (i.e. entering a null password), the block interprets this as a request to cancel out of the PasswordCheck block and the block exits on the Cancel output. The caller would typically be returned to a previous Menu from which the service with the password was selected.

Max Invalid Password Entries Per Session As well as having a maximum number of invalid password entry attempts on each PasswordCheck block, there is an overall limit on the number of such attempts that can be made during a single call. This is to prevent a hacker from repeatedly trying to break through a PasswordCheck block.

If not for this session limit, callers might be allowed a limitless number of attempts to hack a password, because the Cancel output could be used to exit the PasswordCheck block before the block's limit is reached. Re-entering the block, re-starts the invalid count (for the block). When this limit is reached an event is generated and will form part of a Reporter alert or report 1 . The Max Invalid Password Entries Per Session field is set in the Session Profile. The range is 1 to 99. The default is 10.

B SDL Model of PasswordCheck

The PasswordCheck component was modeled as a 4-page SDL specification, given in Figures 15-18 below.

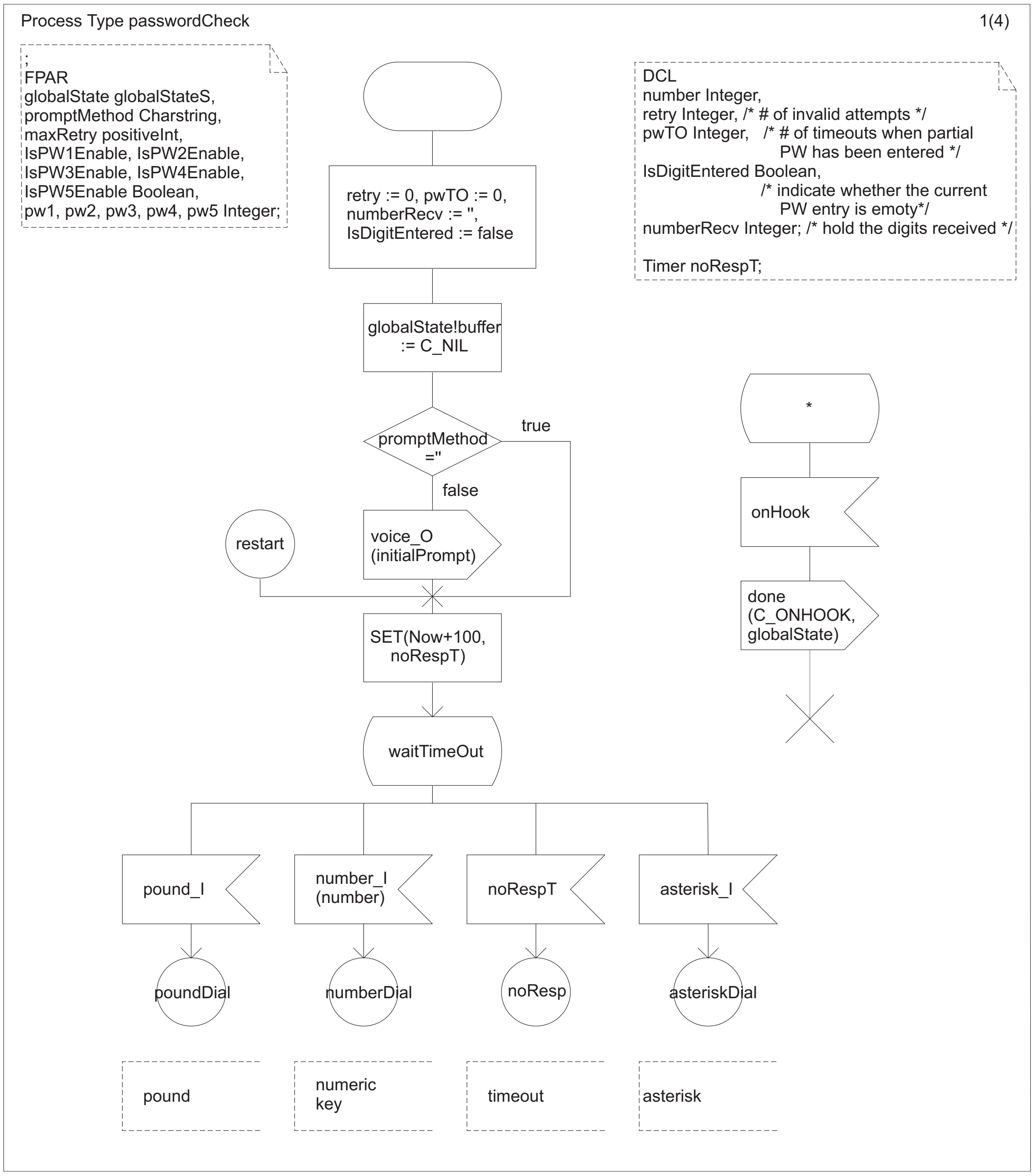


Fig. 15. The SDL model of the PasswordCheck component, page 1.

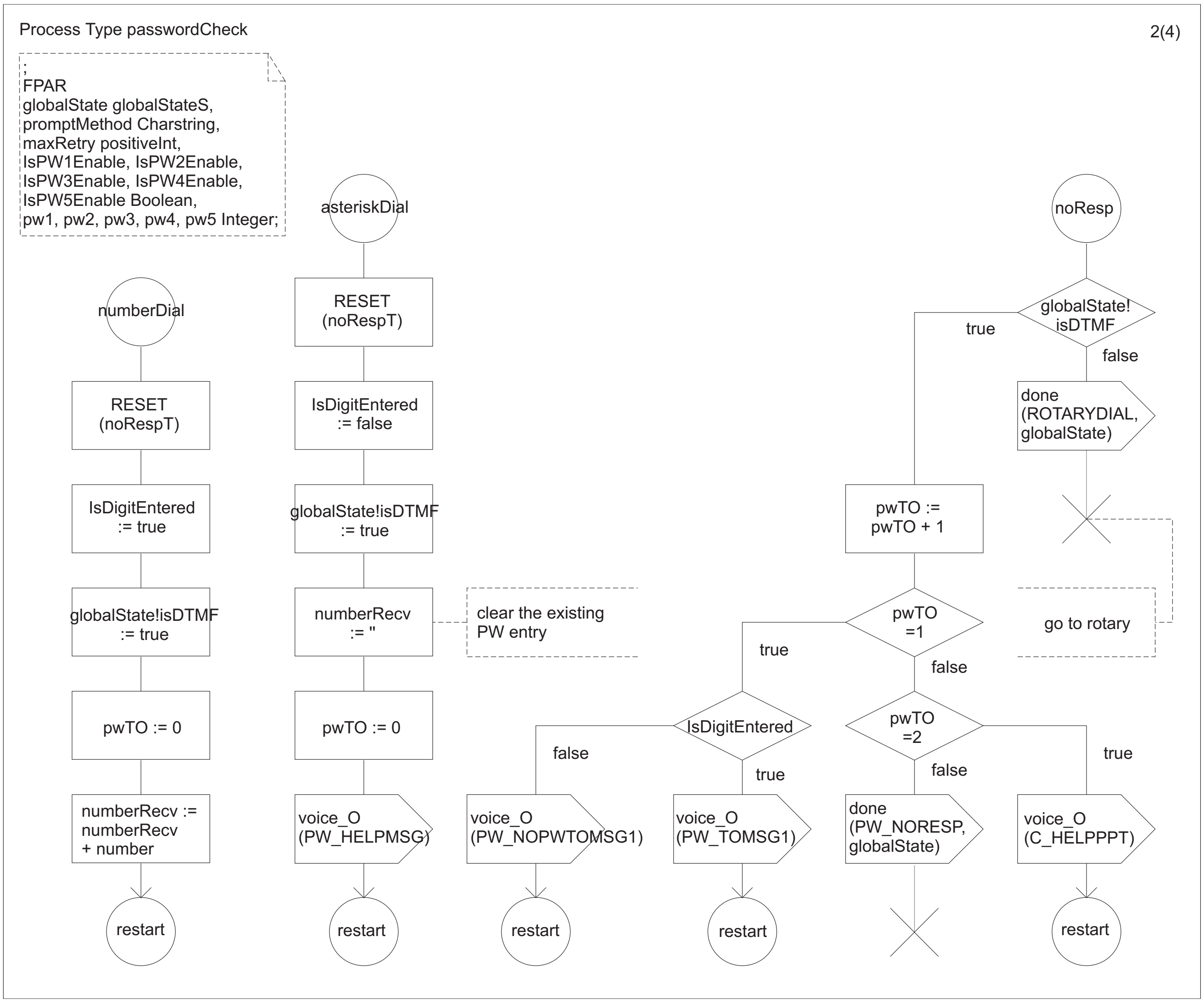


Fig. 16. The SDL model of the PasswordCheck component, page 2.

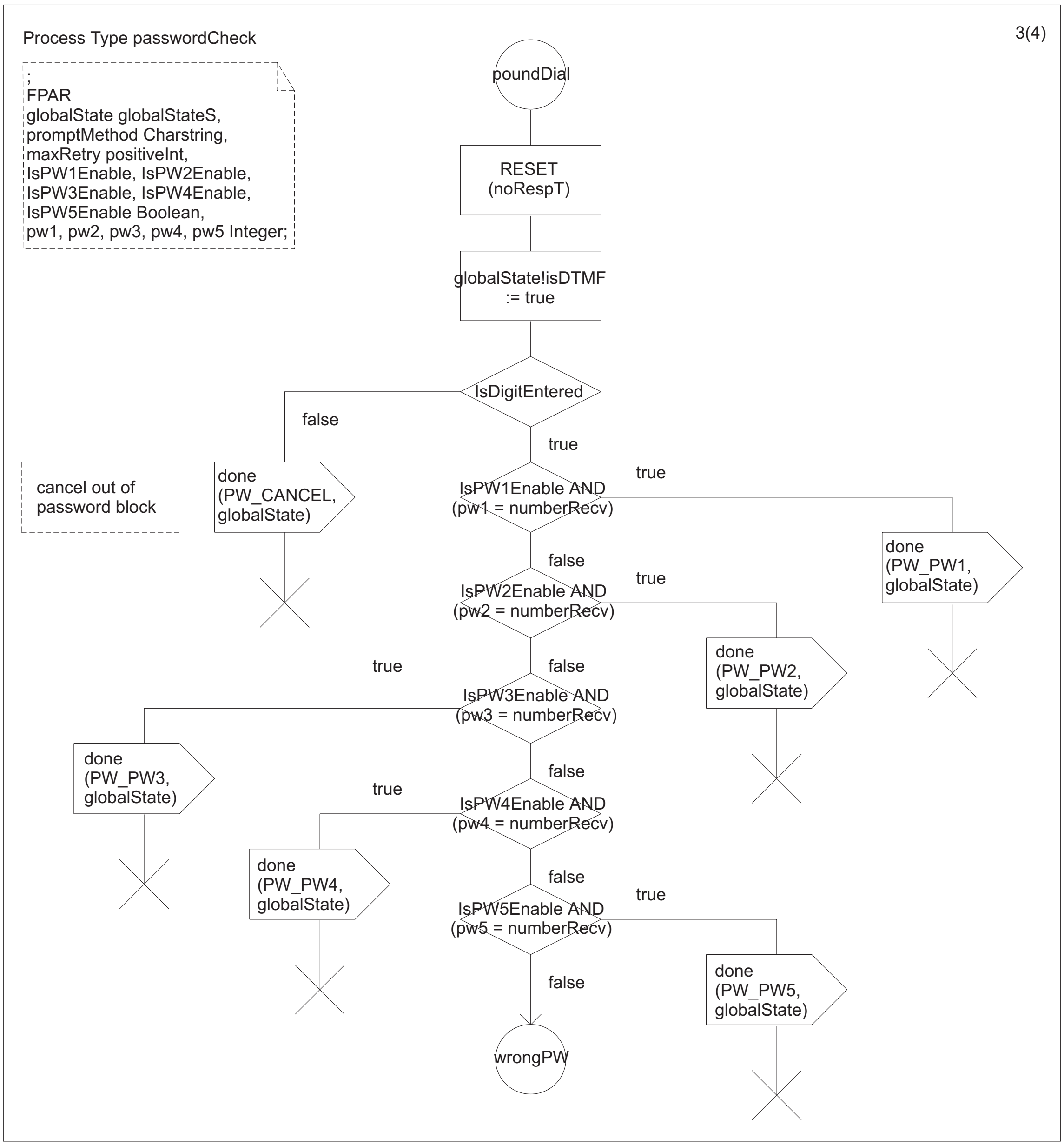


Fig. 17. The SDL model of the PasswordCheck component, page 3.

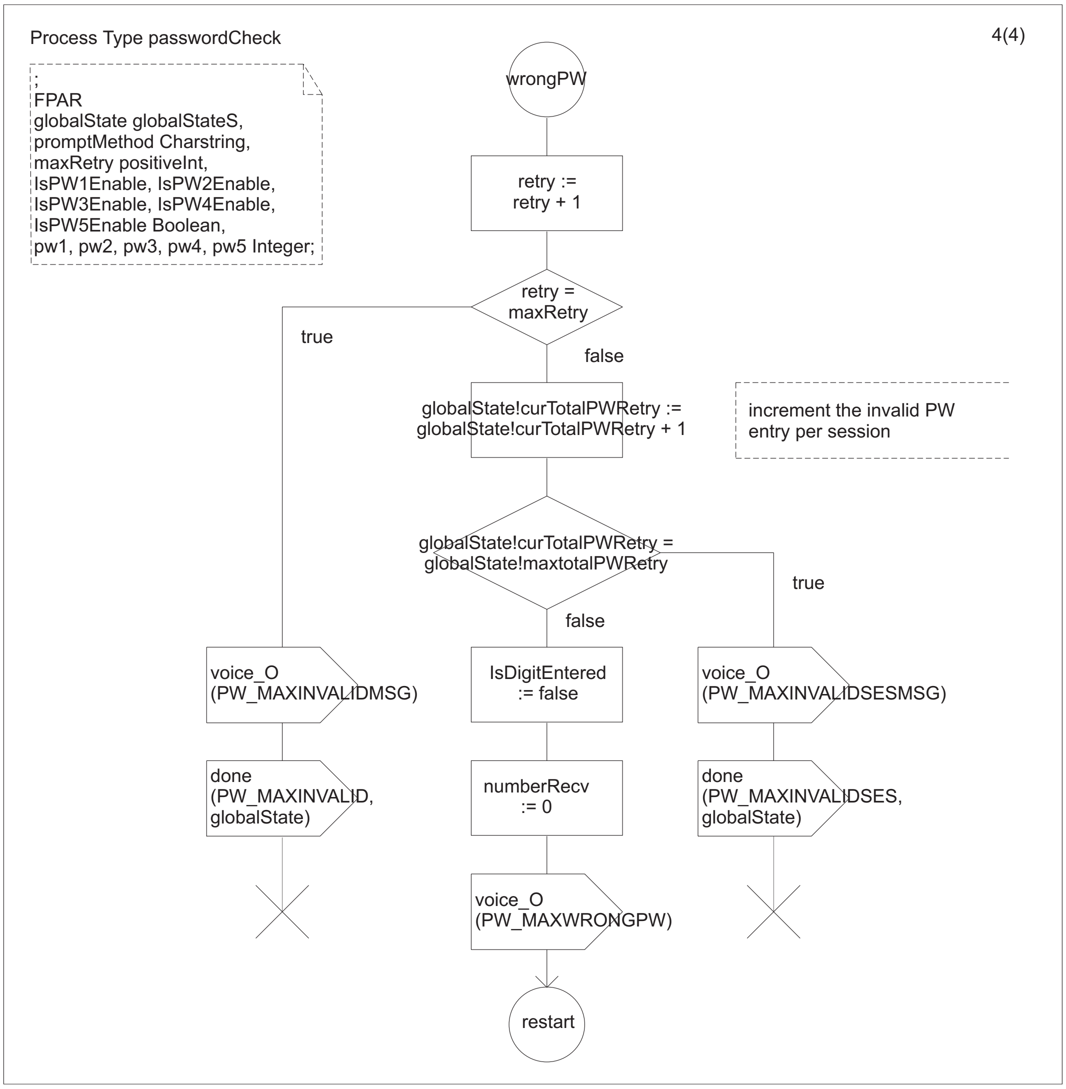


Fig. 18. The SDL model of the PasswordCheck component, page 4.