ELSEVIER

Computer Communications 24 (2001) 374–393

computer communications

www.elsevier.com/locate/comcom

Rapid generation of functional tests using MSCs, SDL and TTCN

R.L. Probert, H. Ural*, A.W. Williams

Department of Computer Science, School of Information Technology and Engineering, University of Ottawa, 150 Louis Pasteur Priv., Ottawa, ON, Canada, KIN 6N5

Received 6 February 1999; accepted 22 May 2000

Abstract

This paper reports the results of a study undertaken to determine the suitability of CASE tools and formal methods for systematic, rapid generation of functional test cases. In particular, the study involves the use of Message Sequence Charts (MSCs) [International Telecommunications Union (ITU-T), Recommendation Z.120: Message Sequence Charts, revised 1996], Specification and Description Language (SDL) [International Telecommunications Union (ITU-T), Recommendation Z.100: Specification and Description Language, revised 1996] and Tree and Tabular Combined Notation (TTCN) [International Standards Organization (ISO), OSI Conformance Testing Methodology and Framework—Part 3: The Tree and Tabular Combined Notation, International Standard 9646-3, 1992] and the use of the Telelogic Tau tool set [Telelogic Tau tool set, version 3.3, Telelogic AB, Malmo Sweden] which supports all three of these languages. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Functional testing; Specification and description language; Tree and tabular combined notation; Message sequence chart; Formal description techniques; Test generation

1. Introduction

The primary objective of the study was to generate functional tests for a telephone switching system from a set of user requirements. In an industrial environment, it is unlikely that a fully complete and correct set of requirements will be available before developing software. Usually, there is an initial set of requirements, and some development work is done using scenarios developed from those requirements as a starting point. As a result of the first phase of development work, modifications are often made to the requirements. Aside from uncovering errors in the first version of the requirements, the consideration of exceptional cases may lead to new scenarios being developed. Thus, an initial prototype may lead to additional or modified requirements.

After a first version of the software is released, developers repeat the whole process for the next version of software, which undoubtedly will have additional features requested by customers. Recommendations for improving the software development process have to take into account that much of industrial software development is inserting new functionality into existing code.

Hence, we wanted to replicate this iterative and incremental process in our study. Fig. 1 shows the process that we followed. We are taking a scenario-based approach, where the scenarios are captured as use cases in Message Sequence Charts (MSCs) [1]. From an initial set of MSCs, we developed an executable SDL model [2]. This allowed us to simulate and validate the model. Simulation allows us to execute the model interactively, and observe that it behaves as expected. Validation allows us to explore the state space of the model to detect properties such as deadlocks, and unspecified message receptions. Building the SDL model also enabled us to produce new scenarios that were added to our set of MSC use cases. Once we were satisfied with the model, we then proceeded to generate abstract TTCN test cases [3] based on the use cases.

A secondary, but equally important objective was to demonstrate the ability of our "fast to text/first to test" approach to dramatically compress the test cycle, and so to improve the time-to-market of the corresponding product. The advantage of our approach is that it is no longer necessary to convince design engineers of the merits of a formal description technique. Instead, the test designers are able to use it at the beginning of the development process so that

* Corresponding author. Tel.: +1-613-562-5800; fax: +1-613-562-5187.
E-mail address: ural@site.uottawa.ca (H. Ural).

0140-3664/01/$ - see front matter © 2001 Elsevier Science B.V. All rights reserved.
PII: S0140-3664(00)00229-2

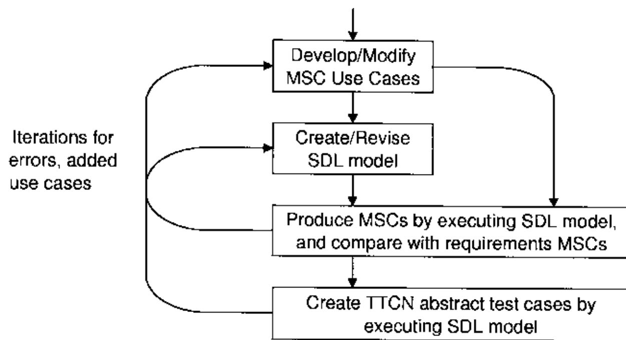


Fig. 1. Iterative and incremental development process.

designers have scenarios coded as use cases in MSC form to validate their designs, often before any code is generated. This provides a cost-effective infrastructure for scenario-directed design.

A tertiary objective was to demonstrate to designers the value of using an FDT-based representation method supported by an industrial-strength (scalable) CASE tool set. Eventually, if designers see the cost-effectiveness of this more formal approach, they will adopt it. We witnessed that testers already know its value.

During this study, functional tests for a basic call, plus calls using four call processing features were derived. Two different models of phone sets were used: a basic set, and a set with an alphanumeric display screen and various lamps and feature keys. By employing the CASE tool coverage analysis features, coverage of all reachable transitions was ensured. Overall, we were able to relate the coverage of all reachable transitions to the generation of high quality functional test cases in a reduced time interval.

As an additional outcome of the study, we have identified recommendations for enhancements to the languages and tools used in the study. We have also developed and applied some productivity measures that support the cost-effectiveness of this approach.

1.1. Related work

Doldi et al. [5] assess the combined use of MSCs, SDL, and TTCN for generation of abstract conformance test suites. Their work focuses on conformance testing. Grabowski et al. [6] provide details on the development and use of the Telelogic "Autolink" tool[4], and its use to validate protocols for ETSI and Ericsson. They show how various features of the tool were driven by needs for particular aspects of validation. In a separate report [7], the results are applied to the testing of the B-ISDN SSCOP protocol, and some productivity results are provided. A report by Schmitt et al. [8] focuses on generating conformance tests using the Telelogic Tau "AUTOLINK" facility, and gives results on its performance. Kerbat et al. [9] provide a corresponding report for the development and use of the TVEDA and TGV tools that are integrated in ObjectGEODE. Kahlouche et al. [10] conducted an experiment to generate conformance TTCN tests for a cache coherency protocol. This study used both MSCs and TTCN, but did not use any formal description technique to capture a test model in the sense that we used SDL. Monkewich [11] proposes the use of a verification and validation process based on SDL that spans the industrial product cycle from design through to field deployment of a protocol-based implementation, although requirements analysis is not included. Additional experience reports for conformance testing of various protocols using MSCs, SDL, and TTCN are available in Cavalli et al. [12], and Pérez et al. [13].

Feijs et al. [14] applied a systematic approach to service testing (using PSF [15]) that considered the effect of a service testing architecture the generation of a TTCN test suite. Most of their work is a manual application of the process, with limited automation between process stages. Anlauf [16] has also tried using TTCN for testing of services, using a manual approach. Fischbeck [17] has used SDL in specification and validation of ISDN layer 3 protocols for narrowband and broadband ISDN. This study determined that SDL provides adequate support for the composition of large specifications, and that there are difficulties to extend signals, data, and transitions in derived specifications. Karoui et al. [18] have investigated how to structure an SDL specification to improve testability. The objective is to enhance the observability and controllability of an implementation, so that it can be more easily checked using data flow testing methodologies.

Our work differs from the above work in that we used TTCN for functional testing instead of conformance testing, and that we have spanned the entire process from requirements analysis through to test case generation by using SDL, MSC, and TTCN, and applied it in an industrial setting. Lai [19] discusses the gap between academic research, and the use of advanced test technologies in the communications industry, and we hope to contribute to closing that gap.

1.2. Organisation of this paper

This paper is organised as follows. In Section 2, we give an overview of the study and comment on the distinction between functional and conformance testing. Next, we describe some of the features of the Telelogic Tool set, as they pertain to our study. We then describe details of the process that we used during the study. In Section 3, we first provide our productivity results as a guide to developing an estimation strategy for the future. Next, we provide an assessment of the languages and methodologies. We also discuss the effects of exceptions, transient states, and race conditions on this process. In Section 4, we comment on our experience with the particular tool set that was used. Finally, in Section 5, we present some conclusions and further work.

2. The study

2.1. Overview of the study

1. We started with system documentation intended for a customer of Mitel. We also had five sample English language test scenarios.
2. From the initial information, we created six MSCs manually (see Appendix A for an example). One MSC use case is for a basic call; four MSC use cases represent "normal" scenarios for the use of each feature of interest: call forward, call back, call transfer, and conference call; and the last MSC use case includes a combination of these features.
3. We then manually developed two SDL models. Both models contain the basic call functionality. One model has feature functionality added, while the second model had set-specific information for a feature-rich phone set. The models are "test models" in the sense that they are not intended to model the entire system implementation. That is, our models were not created with the intention of eventual code generation. Instead, they contain all externally visible system behaviour, plus enough internal behaviour so that the model is executable. See Appendix B for an example.
4. Once the SDL models are developed, they are verified using the Tau SDL simulator. The SDL models are compiled and executed interactively or from scripts. Tau can produce an MSC that records the simulation execution (see Appendix C for an example). We checked that the resulting MSCs were consistent with the original requirements MSCs. We also tried other exceptional scenarios to see if our test model would react properly in these situations.
5. The Tau SDL validator was invoked, to look for situations such as deadlocks, unspecified message receptions, etc. This step is automated.
6. With the verified SDL test models, we performed coverage-based test generation (described in more detail in Section 2.7). The result was a set of test MSCs that achieved transition coverage of each SDL models.
7. Our first method of generating tests was to use the MSCs from step 6 as a guide for interactive test generation using the TTCN link tool. This turned out not to be practical except for the simpler of the two SDL models.
8. We used the Autolink and the set of test MSCs from step 6 to generate TTCN dynamic behaviour scenarios. The MSCs are validated as part of this process.
9. At this point, we examined the TTCN test cases to determine a set of test steps that are common among various test cases. Test step MSCs were selected manually from the MSCs used in step 6, and then entered in the MSC editor using cut and paste. (The results are shown in Appendix D). Then, the Autolink test generation process was rerun to use our library of test steps. The resulting test cases were much more readable and modular (see Appendix E for an example). The modularity of the test steps also allows for substitution for different versions of the feature functionality. For example, a test step that goes offhook and checks for dial tone on a basic phone can be substituted by a test step that checks various lamps and screens on a feature-rich phone. We found that a test step usually corresponds to a single SDL transition from the model, as the transition is executed as a unit in multiple test cases.
10. The Autolink output was imported into the ITEX tool. The TTCN link tool automatically generates a set of declaration tables, and the Autolink output provides the dynamic behaviour and TTCN constraint values.

2.2. Functional vs. conformance testing

In this study, our objective was to generate functional tests rather than conformance tests. Conformance testing typically concentrates on the exchange of protocol messages both for their expected sequence and content. The goal is to verify system interoperability by conformance to a standard. The goal of functional testing is to take a user's view of the system, and check that customer requirements have been met. In particular, the actions and reactions of the system are considered from the user's viewpoint, to ensure that the system behaves as the user expects.

There is a distinction to be made between taking a "customer" viewpoint and an "end user" viewpoint in functional testing. While the direct customer of a telephone switch manufacturer is a company that offers telecommunications services, the end user is someone who makes phone calls. In our project, we took the "end user" view, and considered only the actions of the switch as could be controlled or observed from a phone. If we were taking a "customer" viewpoint, we also would have modelled and checked the behaviour of the operations and maintenance console for the switch.

These goals are reflected in the specific types of actions taken during testing, and reactions observed from the system under test. A functional test action might be to go off hook on a phone set. The corresponding conformance test action would be to construct and send a protocol message to the telephone switch as a result of going off hook. Similarly, a functional testing observation would be to check that the dial tone is heard, and various lamps and displays on the phone are updated. The corresponding conformance testing observation would be to check the exact sequence and data format of messages that are sent to the phone as a result of going off hook.

Functional testing also includes the detection of the presence and absence of persistent events. When taking a phone off hook, the dial tone should not only be started, but it should remain on until the user has dialled the first digit. Then, it should be turned off. We should be able to check

that the dial tone is on at any time during the appropriate interval, and that it is off afterward.

It should be stated that both functional and conformance testing are necessary. Functional testing is more indirect with respect to the system under test, in that we are using a phone to construct the protocol messages. With conformance testing, we would construct our own messages (correct, or not), and send them directly to the system under test. Conformance testing is to detect that the exchange of protocol messages is correct and robust with respect to a standard. Functional testing is to confirm that the system behaves as according to customer or user expectations.

2.3. Overview of the Telelogic Tau tool set

For the purposes of this study, we used the Telelogic Tau tool set, version 3.3. Some of the tools contained within Tau are:

- The ORGANISER, for keeping track of user files.
- SDT, for working with SDL, the Specification and Description Language.
- ITEX, for working with TTCN, the Tree and Tabular Combined Notation.
- An editor for working with Message Sequence Charts (MSCs).

SDT has a number of sub-tools included within it:

- An SDL graphical editor, for creating and modifying diagrams.
- An analyser, which checks the syntax and semantics of the SDL, and converts the SDL to or from the SDL PR (phrase representation) text format.
- Various code generators, for generating C or C++ code.
- A simulator, which provides interactive execution of an SDL model. The user can observe progress through the SDL diagram, or record execution using an MSC.
- A validator, that can perform an exploration of the potential state space of an SDL model. Situations such as deadlock, unspecified message receptions, or values out of range can be detected. The user can also trace a path through the state space and have the path recorded as a TTCN behaviour and constraint description using the "Autolink" facility. This can be passed through to ITEX for generation of a complete TTCN test suite.
- A coverage tool, which can report on transition coverage (that is, coverage of state-input signal combinations), and symbol coverage (that is, coverage of decision branches within transitions), achieved using either the simulator or validator tools.

ITEX includes a TTCN test suite editor, syntax and semantics analyser, a test case execution simulator, and executable test generators.

SDT and ITEX can be linked in several ways:

- A "link" file can be generated by SDT, that ITEX uses to generate all of the declaration tables (for example, signal names, points of control and observation, etc.). This link file can also be used to synchronise SDT and ITEX for stepping interactively through a path in the SDL, and having it recorded as a TTCN test case.
- A path generated by the validator Autolink facility can be merged into a TTCN test suite contained in ITEX.

2.4. Construction of MSC use cases

Before constructing MSC use cases from the given set of scenarios (written in English), a test architecture, which takes into account the nature of the tests and test environment, must be determined. Because we are taking an end user view, functional testing of a telephone switch is necessarily an indirect process. In our study, we determined that the functionality of interest was that the interaction between a phone set and a switch exhibited the correct behaviour when using a phone. Therefore, our test actions will be to take the phone on or off hook, and press keys on the set. Our test observations will consist of observing the lamps, displays, and detecting the tones produced by the phone speaker in the handset.

Another element of indirection comes from the nature of telephone calls. If we are interested in observing that an incoming call works properly on our phone set of interest, another phone must be used to stimulate the switch to produce the incoming call notification. Fig. 2 illustrates these situations.

Eventually, what is described in Fig. 2 as "test tool" commands or responses has to become TTCN send and receive commands to execute tests in the automated execution environment.

The particular environment, for which the test cases are targeted, has a TTCN compiler with an "adaption layer" that contains information about the specific protocol implemented in the system. That is, the adaption layer converts TTCN messages to specific commands that will drive the protocol interface to the system under test. Fig. 3 illustrates this architecture.

The result, from the test case perspective, is that the adaption layer defines the legal set of TTCN messages that we are allowed to use. Therefore, our TTCN test cases must use those same message names and formats to work with the automated test execution environment. Since we are generating the TTCN tests automatically with a tool from an SDL model, this in turn means that the SDL model must also use these exact same message names and formats. Fig. 4 shows this correspondence.

This correspondence goes one step further when MSC use

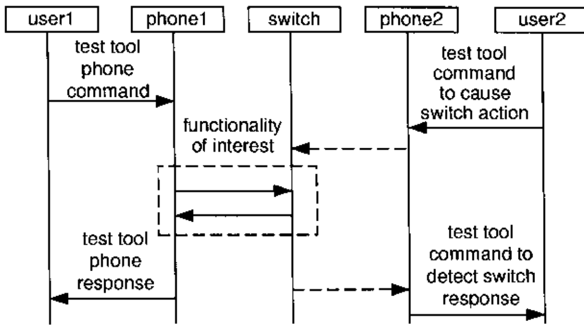


Fig. 2. Mapping the MSC model to the test environment.

cases are used to drive the test generation tool to produce TTCN test cases automatically. The names of the messages in the MSC use cases must also correspond with those in the SDL model, at least at the boundary between the environment and the system.

MSCs were used as a means of capturing scenarios as use cases. MSC use cases are excellent for review purposes, as customers easily understand them. We used the MSC use cases to confirm that we had correctly captured scenarios. The first MSC use case we produced was for a basic normal call. Fig. 5 shows the type of MSC use case produced for a basic normal call. We refer to this type of MSC as a "requirements MSC", as in Appendix A.

Later, we also captured four call processing features, to determine how difficult or easy it is to add functionality incrementally. The MSC use cases that are produced are "system level" MSC use cases in that they capture only the interactions between the system and the environment. For our functional testing viewpoint, this means capturing the interactions between a user and the various phones used in making phone calls. In particular, the automated execution environment would take on the role of all the phones, and therefore we are capturing the interactions between the test environment interface and the user.

Because the system level MSC use case exactly captures the commands that we need to send to the automated test execution environment, it can also be used as a basis to drive the automated execution of TTCN tests. Each message in the MSC use case in Fig. 5 was converted into a TTCN send or receive command by the test generation tool.

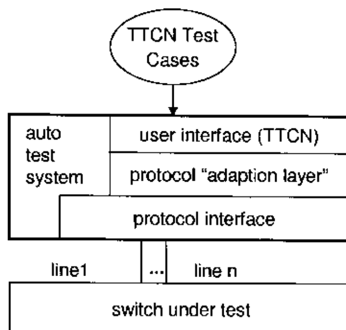


Fig. 3. Test architecture.

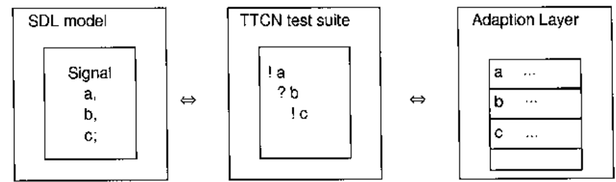


Fig. 4. Determining SDL message names based on the adaption layer.

2.5. Creating the SDL models

In this study, we developed several iterations and versions of SDL models. For testing purposes, the SDL models are not intended to replicate the entire system behaviour of the switch and phone sets. Instead, each model reflects the goals of requirements capture, and of testing those requirements. It is possible that the models we produced could be used as inputs to the detailed design process for further refinements, and to include behaviour that is internal to the system.

It was observed that for functional testing, a model should include:

1. the functionality to be tested;
2. additional behaviour extending to points of control and observation, for test tool commands;
3. enough internal behaviour to execute the model.

The model does not need to include internal behaviour of the system that cannot be observed by test equipment, and is not part of the protocol.

The first version was to produce a model for telephone

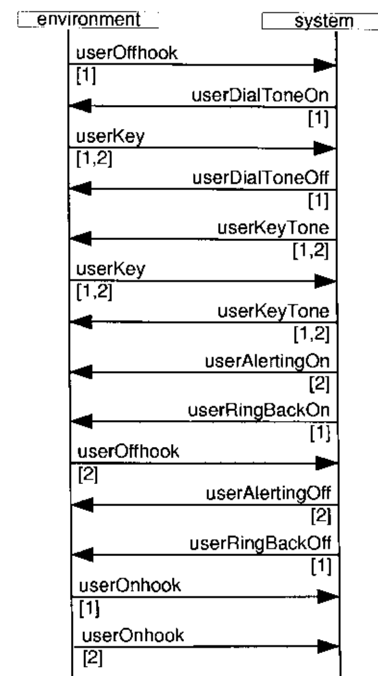


Fig. 5. A system level message sequence chart.

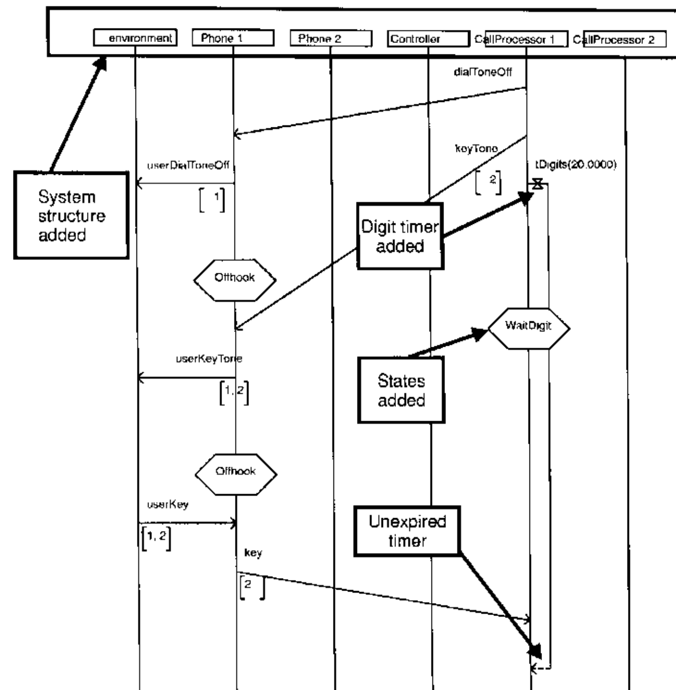


Fig. 6. Differences in an execution MSC.

calls with no additional features. While our original MSC use cases only describe a successful call, once this is captured in the model, we can start to ask "what if…?" questions to examine possible alternatives and exceptions. This is extremely important because it is likely that the software development team will implement normal scenarios correctly. It is the exceptional cases where errors are most likely to be found during testing phases. Therefore, to produce effective, high-yield tests, we must build exceptions into our model. The state-oriented nature of SDL allows the model builder to systematically ensure that, for example, incoming calls have been accounted for in every single state.

Once we had created a model of telephone calls with no extra features, we developed two more versions of SDL test models. One was to determine how difficult or easy it was to add additional call-processing features. In essence, this is adding to the functionality of the telephone switch. Our second additional model was to investigate adding to the functionality of the phone set. That is, we added features such as a display screen, additional function keys, and multiple lamp indicators. This still impacts the functionality of the switch, as the switch has the responsibility of controlling the operation of these additional features on the phone set. See Appendix B for an example of this model.

By using Telelogic Tau's simulated execution facility, the user can interact with an SDL model, and view execution graphically. The user can record the execution as an MSC, and inspect the resulting MSC (henceforth called the execution MSC) for anomalies. The execution MSC can also be compared against the equivalent requirements MSC to see if it is consistent. This allows the model builder to confirm that the model correctly implements the initial requirements. See Appendix C for an example of an MSC produced by simulating the SDL model.

The requirements MSCs and the execution MSCs are not identical though (refer to Appendices A and C). They should be consistent in the exchange of messages between the system and environment. But they can differ because of extra information that results in building the SDL model. The SDL model contains states and these are automatically inserted into the execution MSCs. Timers are incorporated into the SDL model, and their setting, expiry, and cancellation appear in the execution MSCs. Also, we found that what was originally envisaged as two original messages were consolidated into a single message with parameters in the execution MSCs. Finally, the execution MSCs contain architectural information inside the system, including internal messages sent between system components. Fig. 6 shows an example of an execution MSC that highlights differences from the corresponding requirements MSC.

2.6. Validating SDL models using state space exploration

We also checked the SDL models using the state space validation tool in Tau. The validator checks for deadlocks, unspecified receptions, and can determine the consistency of an MSC with the SDL model. The output from the tool for any anomalies consists of a description of the problem

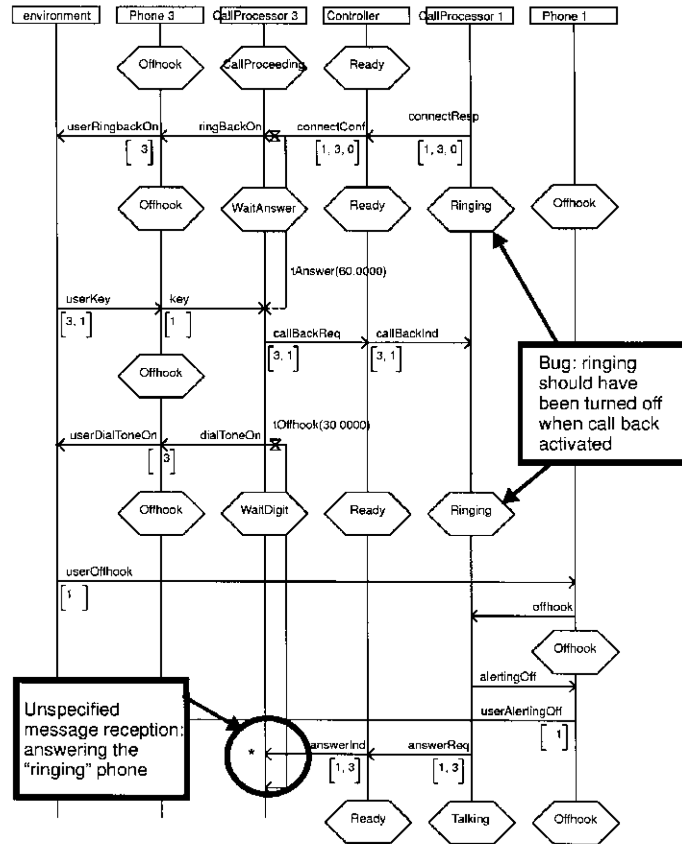


Fig. 7. An MSC trace showing an unspecified reception.

found, and an MSC use case that describes what had happened up to the point when the problem was discovered.

In our study, we ran the validator on our SDL models, and found that an unspecified reception turned out to be an error in our SDL model of the call back feature (illustrated in Fig. 7). The feature is activated after a called party did not answer a call. It seems that we forgot to have the feature activation terminate the ringing of the called phone. The (still ringing) phone was later answered, but the "answer indication" reception was unspecified at the caller.

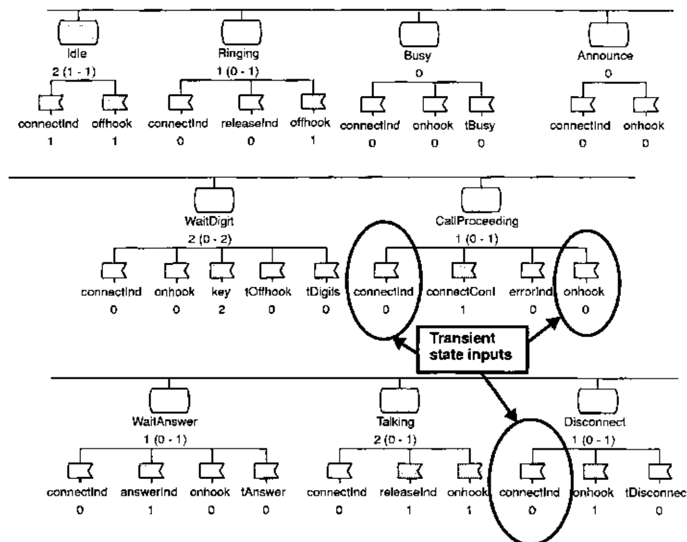


Fig. 8. An SDL transition coverage report.

2.7. Coverage-based test generation

The test cases are constructed using a transition coverage based approach, based on Telelogic Tau's transition coverage tool that counted the number of times that a transition was executed. The tool provides a graphical display of the set of states within the SDL model. The set of possible inputs for each state is shown in a tree format. The inputs are shaded if they have been covered at least once, and are clear if they have not been covered (refer to Fig. 8). The numbers below the SDL input symbols show how many times each transition had been taken.

We started by creating a test case representing a normal call. We used the coverage tool to display the coverage achieved by this scenario. In our model, a normal call achieves about 30% transition coverage.

The strategy was then to choose a "target" state and input combination, and attempt to create a TTCN test case that results in covering this particular transition. Fig. 8 shows an example of a coverage report produced during this process. After the new TTCN test case had been created, its coverage information would be merged with prior results, to get a new view of the current level of coverage. This process was repeated until all reachable transitions were covered (Fig. 9).

While we use transition coverage of the SDL models as a coverage metric, we also ensure that the test for each transition is of a functional nature. That is, the test preamble for a particular transition is chosen to represent how a user would find themselves in a situation where the transition in the model would be executed. The same is true for the postamble. The result is a realistic scenario from the user's viewpoint, and therefore it is a functional test. While transition coverage of the SDL model may not be sufficient for complete functional coverage, it is certainly necessary because our model is originally based on user requirements.

The coverage-based approach gave us a series of test cases that concentrates on exceptional cases. Of the eighteen test cases we generated for call processing with no features, there are five cases where timers expire, three cases where the caller aborts before connecting, one call to a non-existent number, and eight cases where incoming calls are rejected with a busy signal.

The coverage-based approach also highlights situations where it may not be possible to create a reliable test case, due to transient states or race conditions.

A transient state is one that, from an external view of the system, lasts for a very short interval. In our SDL model, "Call Proceeding," is a transient state. This state occurs just after a calling party has dialled the final digit. The state lasts only as long as the silent period between dialling the last digit, and hearing ring back, a busy tone, or the invalid tone. A connection request is sent to find out if the called party is free or not. In a phone system that includes routing outside a switch, this state may last for a substantial amount of time. However, in a system consisting of extension lines on a local switch, the time interval is extremely short.

Nevertheless, this is a distinct SDL state, and therefore the usual transitions handling the user hanging up, and an incoming call, should be considered as possible. Causing this to happen would require extremely fine timing tolerances on the part of the test environment.

normalCall

Test Case Name	: normalCall
Group	:
Purpose	:
Configuration	:
Default	: OtherwiseFail
Comments	:
Selection Ref	:
Description	:

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		user ! userOffhook	offhook_1		
2		user ? userDialToneOn	dialToneOn_1		
3		user ! userKey	key_1_2		
4		user ? userDialToneOff	dialToneOff_1		
5		user ? userKeyTone	keyTone_1_2		
6		user ! userKey	key_1_2		
7		user ? userKeyTone	keyTone_1_2		
8		user ? userAlertingOn	alertingOn_2		
9		user ? userRingbackOn	ringbackOn_1		
10		user ! userOffhook	offhook_2		
11		user ? userAlertingOff	alertingOff_2		
12		user ? userRingbackOff	ringbackOff_1		
13		user ! userOnhook	onhook_2		
14		user ! userOnhook	onhook_1	PASS	
15		user ? userRingbackOff	ringbackOff_1	INCONC	
16		user ? userRingbackOn	ringbackOn_1	INCONC	
17		user ? userAnnounceOn	announceOn_1	INCONC	
18		user ? userAlertingOn	alertingOn_2	INCONC	
19		user ? userDialToneOn	dialToneOn_2	INCONC	

Fig. 9. An automatically generated TTCN test case.

Task Name	Duration (person-days)
Basic call (BC) MSC	12
BC SDL iteration 1	12
BC SDL iteration 2	16
BC SDL iteration 3	3
BC SDL iteration 4	10
BC SDL integrate features	2
BC TTCN normal call	1
BC TTCN iteration 2	3
BC TTCN modular	5
Display phone (DP) MSC	4
DP SDL	11
DP TTCN iteration 1	0.5
DP TTCN modular	10
All feature MSCs	15
Call Forward SDL	5
Call Soft Hold SDL	7
Call Transfer SDL	5
Conference Call SDL	6
Call Back SDL	5
All feature TTCN test cases	22

Fig. 10. Productivity results.

The other type of "difficult" transition is a race condition. This is often the result of two messages crossing each other between the two ends of a call. For example, suppose both parties hang up simultaneously. The release messages may pass in transit, or one or the other might arrive first. It is possible that a call processor might receive a release request in an idle state. If we allow that a release request might be delayed in transit, for whatever reason, the other party may have already started a new call and could conceivably be in any state. Some of the race conditions could be produced using the SDT simulator, by sending multiple messages from the environment before allowing the model to proceed with execution.

An abstract test case can be created for a situation involving a transient state or a race condition. However, running such test cases in practice is usually infeasible. Extremely fine tolerances for the timing and control of the test equipment would be required because such test cases involve a particular sequence of messages that must be sent or received in a particular order within a short interval. This interval may not have observable cues to define the suitable range over which messages can be sent. Furthermore, it is desirable to have test cases that produce consistent test verdicts when executed repeatedly. Test cases requiring tolerances close to the speed of the test equipment may instead produce variable results over several execution runs. In test plan documentation, it is useful to note that an abstract test case for a transient state or race condition was considered (for completeness), but that an executable test case could not be created. Therefore, a test manager can expect that all executable test cases can be run.

Creating reliable test cases for an automated execution environment that involves either race conditions or transient states is extremely difficult, if not impossible. It requires extremely fine tolerance for timing and control of the test equipment that is used.

3. Results of the study

3.1. Productivity results

Before presenting our productivity results, it is necessary to describe how many people were involved and how much product, tool, and methodology knowledge they had. We had one product expert working with two students. The product expert had recently taken courses on the tools and methods, and one of the students learned these while doing the project. The other had prior experience with the use of the tools and methodology, and with telephone switch testing.

During the three months of the project, the students were able to devote most of their time for two months to the project, but during one month, the students were concentrating on other academic work. Fig. 10 shows a project time line containing the time taken for various elements of the project (BC = basic call and DP = display phone).

The real time length of the project was 78 days, which includes five days for project planning, and 23 days for project reporting (including metrics collection).

For test case generation using the coverage-based approach, we have some additional statistics. We found that it takes a larger amount of time to create the first test case, but the time interval becomes progressively shorter. The first test case took 40 min to produce. The shortest interval to produce an additional test case was only 2 min. On average, it took about 15 min per test case. The test case generation was done over a three-day period, as noted in the above table.

For both the basic call, and the display phone, 18 test cases were produced to meet the coverage criteria. For the SDL that integrated all of the features, 51 test cases were produced to meet the coverage criteria.

We found that while producing a TTCN test case using the Tau tool is quick, adding test steps (a manual process) is much more time-consuming. We felt the effort was justified to produce test cases that were far more readable, and modular.

These productivity results compare very favourably with the internal Mitel average in terms of time. In addition, the test cases have a measurable extent of coverage. This measurement capability enabled a systematic approach to test development and project management.

3.2. Suitability of the standard languages: MSCs, SDL, and TTCN

In general, we found that all three of the standard

languages were suitable to meet our goals. However, the following issues arose in our study.

3.3. Message sequence charts

1. *Enforced system structure.* By its nature, an MSC may enforce a system structure. This is the result of having a set of defined actors among which messages are exchanged. One could hide the internal structure by having an MSC that describes only the interaction between the system and the environment. The validator uses this type of MSC to generate TTCN test cases. However, this MSC style can appear to be quite different visually from an MSC that includes the internal structure. This leads to the next issue.

2. *Determining consistency of MSCs.* Answering the question, "Are these two MSCs consistent?" turns out to be non-trivial. If two MSCs have different levels of abstraction, it can be difficult to determine if they are consistent. We found that this step could not be done automatically, and therefore had to be done by manual inspection.

In situations where some message reordering is permissible, two MSCs may be consistent but not identical. The use of the MSC 96 "co-region" feature could be used to address this issue.

One important type of comparison is to compare a requirements MSC with an execution MSC produced during the simulated execution of the SDL model. However, these MSCs can appear to be quite different. We found that in the process of creating an SDL model, we made various design decisions that were not anticipated at the requirements level. For example, we initially assumed that there might be separately named messages to either accept or reject a connection request. In the SDL model, we decided to define a single connection response message, and include a parameter that indicated whether or not a connection was accepted (and if not, why).

Another issue arises if states are included in the requirements MSC. If they are, the set of states may change as a result of design decisions. For example, we had assumed that there would be two distinct states: "wait for first digit", and "wait for subsequent digit". The distinction is that the dial tone has to be turned off between these two states. However, in the SDL model, we decided to combine the two states, and use a counter to keep track of how many digits had been dialled. This counter was useful in another context, namely, determining if the final digit had been dialled or not. However, the execution MSC now has a different set of states from the requirements MSC.

It is also likely that exceptional conditions, in particular, time-outs, are also not included in the requirements MSCs. The starting, expiry, and cancellation of timers are also actions that may be included in the execution MSCs but not in the original requirements MSCs.

Fig. 6 shows examples of the differences that we have just described. The figure shows an execution MSC, and highlights the attributes that are different from a requirements-style MSC. In particular, states have been added, and timers are started and cancelled. Also, where a requirements MSC would most likely have two actors ("environment" and "system"), the illustrated execution MSC shows the system substructure.

It is apparent that the two types of MSCs could have considerable variations between them, so determining if the execution MSC represents a correct implementation of a requirements MSC is non-trivial, and must be done using a manual inspection process.

3.4. Specification and description language

We also found that SDL is quite suitable for the capture of our test models. While it takes longer to learn SDL to the point where one can produce a syntactically and semantically correct design, the basics of SDL are easily grasped for review purposes.

The issues related to SDL that were raised during our study were:

1. *Forced ordering of actions and outputs within a transition.* Inside an SDL transition, there could be a number of output messages sent. While these messages may be sent to various locations, it is possible that several messages may be sent to the same process. On a feature-rich phone set, a single input event may result in multiple actions.

For example, suppose that a single transition when going off hook has two outputs to the **same** phone: one to turn on a lamp, and the other to update a screen. When a TTCN test case is generated automatically from this transition, the result is:

```
! offhook
   ? lamp on
      ? screen update
```

The difficulty is that the strictly sequential nature of a transition in SDL imposes an ordering on the sending of the messages to the phone set. For user oriented functional testing, the order is irrelevant, because the user is only interested that both results happen within a suitable interval. When TTCN is generated from this transition, it is assumed that the two messages would arrive in the order indicated.

However, for the purpose of the test specification, we really do not want to impose a specific ordering on these events. We do not want to fail the test case if these two events are detected in the opposite order. All we want to do is to detect that after we go offhook, the two events happen in any order.

Furthermore, if the specification is being used as a basis for the detailed design, and eventual implementation, the ordering is being imposed as an unnecessary constraint for the designer. There should be a way to indicate that these two events must happen during the transition, but that the designer is free to choose the order. This would allow the

designer to optimise the implementation for performance, memory, etc., and still conform to the specification.

2. *Single input queues for an SDL process.* The semantics for an SDL model are that each process has a single input queue. Therefore, if a message with the same name can be received from two locations, there is no way to determine the origin of the message. One can use SDL process identifiers, or add message parameters to resolve this difficulty.

3. *Signal routes on which a signal can be both sent and received.* There is a signal routing issue, when a signal route can both send and receive a signal with the same name. If an SDL process sends this signal, the user would expect it to be sent over the associated signal route. This does not happen. Instead, the process winds up sending the signal to itself. This is usually not what is intended. The only way to send the signal out of the process and along the signal route is to use a TO or VIA clause.

4. *Signal routing clauses: TO versus VIA.* The set of connections to the environment determines the set of points of control and observation to the system. Care has to be taken so that these connections to the environment are consistent with how TTCN interprets them. SDL allows two ways for directing a signal. The user may specify a particular route or channel, using a VIA clause. Alternatively, the user may select a particular SDL process as a destination by using a TO clause. However, TTCN only allows signals to be sent over defined points of control and observation (PCOs). A PCO is the endpoint of an SDL channel that connects to the environment. Therefore a TTCN test case can only specify a channel, and not a process. If the SDL model can only function in simulation by sending messages using a TO clause, there is no corresponding TTCN construct.

3.5. Tree and tabular combined notation

The use of TTCN of express functional tests was an unusual use of TTCN, which has primarily been used for conformance testing. Recently, ITU has opened several new questions involving extensions to the definition, capabilities, and applicability of TTCN and Concurrent TTCN, including the ability to specify functional testing and performance testing. We see our work as contributing to this work on extending and improving upon the current versions of TTCN.

The issues with TTCN for the purpose of constructing functional test suite are:

1. *Forced ordering of events.* In conjunction with SDL's forced ordering of events, it may also be that the test environment may not detect events in the same order as they occur at the system. For example, if a phone has to have a lamp turned on, and the screen display updated, it may be very quick to detect the state of the lamp (on, off, or flashing). Determining and checking the contents of a screen display may be more time consuming. Therefore, recognition of events at the test environment may differ in order from their initiation by the system. For functional testing, TTCN needs to have a provision to detect a concurrent set of events. For example, there may be four elements of a phone to be verified: the audible tone, the main lamp, the screen display, and the line lamp. It should be possible to declare that checking the current state of these events is not dependent on any particular order. Instead, they should all occur within a specified time after the event that initiated them. Specifying all possible orderings is generally not reasonable, since this grows at a factorial rate (e.g., five concurrent events have 120 possible orderings).

SDL considerations aside, TTCN does allow us to explicitly provide a number of alternative event orderings. But, we found that on the phone set with many display features, there were typically four update messages for each action performed. Because the number of orderings is factorial, there are 24 possible orderings of four events. Specifying each of these every time is not a usable solution.

There needs to be a mechanism in both TTCN and SDL to specify the following:

```
! offhook
   ? screen | main lamp | line lamp | dial tone
```

where the interpretation is that the events separated by vertical bars could happen in any order, but without intervening messages and before subsequent actions.

2. *Generation of TTCN alternative paths.* This point is related to the last one in that the root cause is that we want to leave some message orderings unspecified because a group of them could happen in any order. In this case, we are discussing the process of generating tests, and how to account for such unordered events.

We are using MSCs to guide the test generator, but the MSC contains only a single ordering of possible events (although the co-region feature of MSC-96 allows multiple orderings). Using the example from the previous section, if the MSC happened to have a particular ordering of the messages to a phone's display screen, the two lamps, and the tone generator, the test generator needs to recognise that other alternative orderings are possible. The test generator typically adds a TTCN "otherwise" clause that would cause the test case to fail if any messages other than the one desired arrive at any particular time. But, if there are three other messages that could arrive, and do not represent a violation of desired behaviour, then this needs to be considered by the test generator. The TTCN test case that results should ideally add alternatives that will also lead to a pass verdict.

The tool we were using overcame part of this difficulty by using the state space search mechanism for identifying alternatives. However, instead of taking each alternative branch to its conclusion, if an alternative message arrived, the verdict of the test case was immediately declared to be "inconclusive". The result is that we would still have to generate separate test cases for each possible ordering.

But, at least with this approach, valid behaviour would not result in a test case assigning a verdict of fail.

3. *Persistent events.* There is a question of exactly what events we should be detecting for the purposes of functional testing. In our SDL model, we have messages such as "dialToneOn" and "dialToneOff." The intention is that after a "dialToneOn" message occurs, the dial tone does remain on, and that test equipment can now detect its presence. Similarly, after a "dialToneOff" message, we are assuming that the test equipment can now detect the absence of the dial tone.

The use of "-On" and "-Off" messages allows for interpretation as non-persistent message receptions, or the start of a persistent state. TTCN has the model of non-persistent message receptions, so our SDL model can be interpreted as consistent with this approach. However, the detection of persistent states is an element of functional testing that TTCN has to be able to handle.

4. Experience with Telelogic Tau

Tau provides two methods of generating TTCN tests. The "Autolink" tool generates TTCN dynamic behaviour based on an MSC scenario. The "TTCN link" tool allows the user to interactively act as the environment by sending messages to the system; the tool determines the system responses.

The TTCN link tool also generates all of the TTCN declarations that can be determined from the SDL model. This tool must be used for this purpose for either method of test generation. This type of information includes the names of messages, the number and type of fields for each message, etc.

4.1. Autolink, and the SDT validator

The Autolink tool is incorporated within the SDT validator tool. It allows the user to generate the dynamic behaviour part of a TTCN test case. The behaviour is directed by a "system-level" MSC. Essentially, a system-level MSC is the same as a requirements-level MSC. The particular restriction for a system-level MSC is that there are only actors for the SDL environment, and for the SDL system. There should be no system substructure.

Generating the dynamic behaviour is a two step process. The first step is to verify that the MSC is consistent with the SDL system. This is done using a state space exploration of the system. Once the MSC has been verified, then the TTCN dynamic behaviour and constraints are generated. If there are valid alternatives at any point in the dynamic behaviour description, they are added as alternate TTCN branches with an "inconclusive" verdict. A default table is set up so that any invalid behaviour inconsistent with the MSC will produce a "fail" verdict. Only behaviour corresponding exactly to the MSC will result in a "pass" verdict.

The following is the type of test case produced by the Autolink tool, with the state space exploration turned on:

```
+ initialPartOfNormalPhoneCall
  phone1 ! keyFinalDigit
    phone1 ? ringbackOn
      phone2 ? alertingOn
        + restOfTestCase    PASS
    phone2 ? alertingOn     INC
```

In this case, the requirements MSC assumes that the caller will hear the ringing tone before the called party's phone rings. However, it is equally possible that the called party's phone may ring first. Because this is not an exact correspondence to the MSC, the alternative is flagged as "inconclusive".

In using the validator, the most difficult aspect was in running state space exploration. There are three kinds of explorations available: exhaustive, bit-state, and random walk. It turns out that exhaustive exploration is not practical in most cases (see [20] or [21]). The random walk is the user-friendliest. It asks how many random walks the user wishes to take through the state space, and how many steps (i.e., the search depth) to take in each walk. We found the bit-state exploration the most difficult to use. This type of exploration comes with a large number of parameters that affect the size of the state space. Despite a good deal of experimentation, we were unable to find a set of parameters that would result in a state space exploration that terminated in a reasonable amount of time, which we arbitrarily decided to be 10 min. We could turn the state space exploration off during the test generation phase, which resulted in creating a test path that exactly mirrors the MSC without considering valid alternatives.

If the state space exploration is turned off, then the following test case results:

```
+ initialPartOfNormalPhoneCall
  phone1 ! KeyFinalDigit
    phone1 ? ringbackOn
      phone2 ? alertingOn
        + restOfTestCase    PASS
```

In this case, only behaviour corresponding exactly to the MSC will be put into the test case. Since the default ?OTHERWISE statement is associated with a "fail" verdict, a valid alternative may cause an unwanted test case failure. On the other hand, this test case is generated instantly.

It also turned out that our state space exploration settings prevented the generation of test cases that involved transient states or race conditions. While these sorts of tests are unreliable in an actual test environment, we were still forced to leave these out of our test suite.

If one is running the validator for the purpose of checking the SDL specification, we were more tolerant of running explorations for long periods of time. The results were impressive: we found five bugs in the SDL specification

that had already been simulated extensively. The validator searches the state space for deadlocks and unspecified receptions. We did not have any deadlocks (having timers in our SDL model took care of that), but we did have a number of unspecified receptions. In one case, this was due to the omission of an action that should have been performed earlier.

We also found several situations that sparked some debate: the SDL environment could send an offhook message while a phone was in an offhook state. We had no message reception specified, since with a working telephone, one cannot produce this effect. However, we did not want to go as far as calling this an impossible situation, in case there might be some sort of equipment defect that might send two offhook messages without an intervening onhook message. As a tester, one must be sceptical of claims that a situation will never happen.

The validator provides a navigation tool that allows one to interactively explore the state space. We found this was a user-friendly way to interact with the SDL specification. Care must be taken with defining the set of messages that the environment is allowed to send. For example, if a message carries two parameters, then for the purposes of the validator, different values for the parameters—and different combinations of values—represent messages that are considered to be different. The greater the number of message and parameter combinations, the larger the resulting state space will be.

When a state space exploration is run, and anomalies are found, a set of reports is generated. The user can double-click on a report indication to produce an MSC that illustrates how the problem was detected. This is extremely helpful for identifying and correcting problems.

Overall, we were impressed with the capabilities of the validator. The only difficulty in using it is determining an appropriate set of state space exploration parameters, especially to balance the extent of the exploration versus the time required to complete the validation.

4.2. The ITEX TTCN link

The purpose of this facility is twofold. The first is for the TTCN tool ITEX to read the static system architecture from the SDL model. The result is that ITEX can automatically generate the part of the TTCN test suite relating to the static system architecture. The second purpose is to establish a communication link between the simulated execution of an SDL model, and the interactive test case generator. This allows the user to execute the SDL model interactively, and record the steps taken as a TTCN test case.

In the first mode, the user can generate automatically the set of TTCN declaration tables from the SDL specification. These include definitions of points of control and observation (PCOs) (in SDL, these are connections to the environment), and the definition of protocol data units (PDUs), which are taken from the SDL signal definitions. This stage is necessary, no matter which method is used to generate the behaviour descriptions.

The user can also use the link to semi-automatically generate TTCN behaviour descriptions. The SDL model is synchronised with ITEX to keep the SDL model in the equivalent state. Once synchronisation occurs, the user can interactively supply input from the environment, and the link will automatically determine the set of valid responses. The advantage is that the alternatives are generated without the state space exploration. As long as synchronisation is maintained the alternatives are generated rapidly. The disadvantage is that synchronisation is either slow (it needs to be repeated if the user changes anything, including converting the constraint names to something meaningful) or impossible.

The SDT-ITEX link produced the following type of test case:

```
+ initialPartOfNormalPhoneCall
  phone1 ! keyFinalDigit
    phone1 ? ringbackOn
      phone2 ? alertingOn
        + restOfTestCase   PASS
    phone2 ? alertingOn
      phone1 ? ringbackOn
        + restOfTestCase   PASS
```

This method produces the "best" test case of the three methods, in the sense that all valid branches are explored to their ultimate end. However, we found that the TTCN link cannot handle "large" SDL models. For large models, the Autolink method had to be used instead.

4.3. Comments on the two test generation methods

Of the two test generation methods, the Autolink method appears to be superior, because it could handle larger models beyond the capability of the TTCN link. Even without the state space exploration, the Autolink can still generate useful test cases extremely quickly, although the quality of the test cases is improved when the state space exploration is on. However, the TTCN link is still required for generating the declaration tables. For this purpose only, we did not reach any limitations on the size of the SDL model.

There is one item that should be added to the test cases generated by either method. When a path goes through a state space node where a timer expires, the interval for the timer does not make its way into the test case. For example, in a test case where the user goes offhook, and expects to eventually hear an announcement, the behaviour appears as follows:

```
phone1 ! offhook
  phone1 ? dialToneOn
    phone1 ? dialToneOff
      phone1 ? announcementOn
```

Between the line where the dial tone is on, and the dial tone is off, the offhook timer is expected to expire. There is no indication in the test case that we are waiting for a timer, which could be approximately 30–60 s long. Furthermore, the automatic default table states that if any message is not received before a time-out, the test case should be terminated with an inconclusive verdict. (This is the default, but it can be easily changed to "fail".) Furthermore, if the dial tone goes off **before** we expect, this would not be caught in this test case. A test case timer should be inserted for any SDL timer expiration, but this is included in neither the Autolink, nor in the TTCN link.

4.4. From abstract to executable tests

The scope of our study was to produce only abstract TTCN test cases. However, we did try to keep in mind that the eventual goal is to produce executable tests. In particular, we tried to ensure that we did not do anything that would knowingly increase the difficulty of converting the abstract tests to executable tests, for an automated environment where the TTCN can be compiled and executed. In particular, we were concerned about the actual test commands available in the test execution environment. A test execution system has a restricted set of messages that can be sent or received, depending on the system or protocol to be tested. From the point of view of test creation in TTCN, the effect is to fix in advance the specific set of messages that are allowed.

Therefore, if we propose to generate the TTCN test cases directly from the SDL specification, it follows that the SDL specification must use the exact names for signals that are defined by the actual test environment. Alternatively, one could create a text editing script to substitute the actual message names for those in the SDL model. This would be necessary for the TTCN compiler on the test environment to accept the test suite.

There are also the issues of assigning (using our example) specific phone lines in place of "phone1" etc. This not only related to TTCN points of control or observation. In dialling a phone call, the test case must use the actual digits to call the other phone. The dial command has to substitute, for example, "2102" (an actual extension number) in place of "2" (an abstract phone number). Furthermore, the phone display would read "2101 CALLING" instead of "1 CALLING". We used the generic command "displayDateTime" for an idle phone that displays the current date and time. In the actual test environment, we would want to check that the actual correct date and time are displayed.

While the actual conversion of the abstract test suite to an executable test suite was outside the scope of this study, we did keep these potential issues in mind.

5. Conclusions

Based on our experience with this project, we have developed a number of recommendations:

1. Our approach provides a means of considerably enhancing the productivity of test design engineers. We were able to generate a correct, value-added, TTCN test case every 15 min, which is considerably faster than with a manual approach.
2. Our approach provides a systematic means of improving the quality of designs. Using the state space exploration found a bug in the functionality in a model that had previously undergone extensive simulation. More significantly, this bug could have been found long before detailed design, and coding.
3. The process should be integrated with the design process, so that both designers and testers can take advantage of having an executable model at an early stage.
4. The correct metrics framework for our project only became clear towards the end of the project. We believe that our experience will now allow us to define better metrics that are usable throughout the process.

A direction we are pursuing is to incorporate an object model into this process. We noted earlier that MSCs impose a structure on the system being described. In the software development process, it is the step of developing an object model that results in the system structure needed by the MSCs. In either a legacy environment, or when commercial off-the-shelf components are used, the system structure will be strongly influenced by the objects that are already available. At the same time, the MSCs are an excellent means of capturing interactions between objects. The integration of an object model would be helpful not only in a technical sense, but would also bring the design community into the process. There are benefits for the use of an object model for test design, which are similar to those for software design. When design components are assembled, corresponding test components can be assembled as well to keep the test suite current. An object model also allows easier modification of test suites as functionality is added, modified, or removed in subsequent iterations of the development process.

Acknowledgements

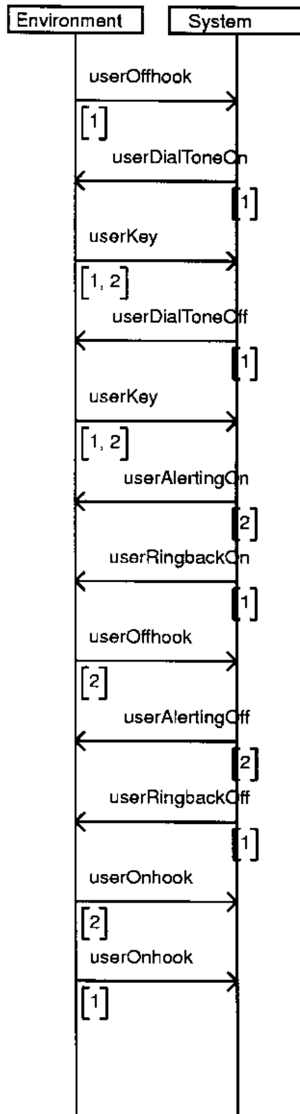
The authors would like to thank the following people for their contributions during this study: Richard Plackowski, Kelvin Steeden, Daisy Fung, and Peter Perry at Mitel; Gregor v. Bochmann, Jun Li, and Rodd Lamarche at the University of Ottawa. Thanks also to Louise Desrochers for administrative support, and to members of the Telecommunications Software Engineering Research Group at the University of Ottawa for their feedback on a preliminary version of this material.

The authors gratefully acknowledge the valuable comments of the anonymous referees; and support for this work from Communications and Information Technology Ontario (CITO), and Mitel Corporation.

Appendix A. Requirement MSC

The following is an example of a “requirements” MSC. The main property of this MSC is that the only actors are the environment and the system. There is no system substructure.

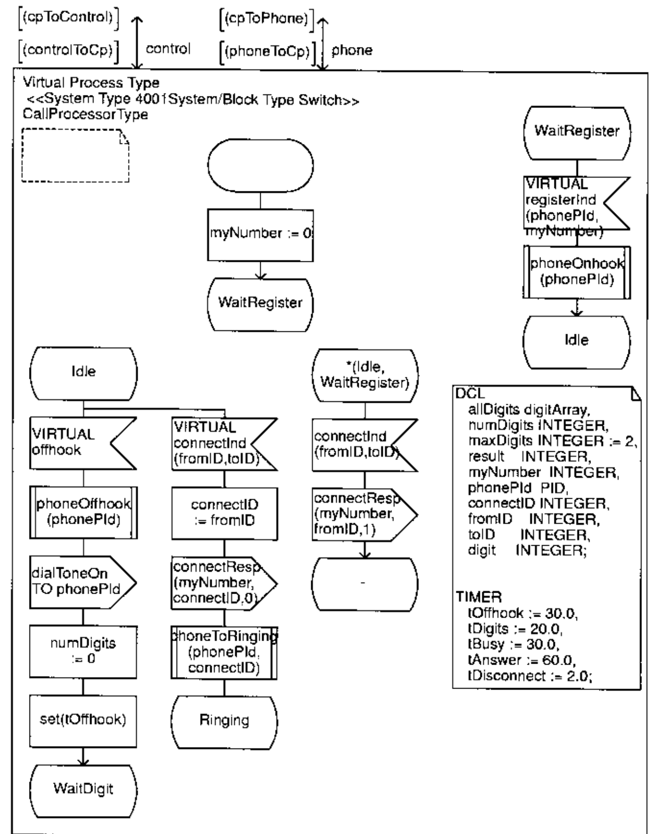
For brevity, it is assumed in this MSC that there are only two digits in a phone extension number. The first parameter refers to the phones; phone 1 has the dialling number ‘11’ and phone two has the dialling number ‘22’.

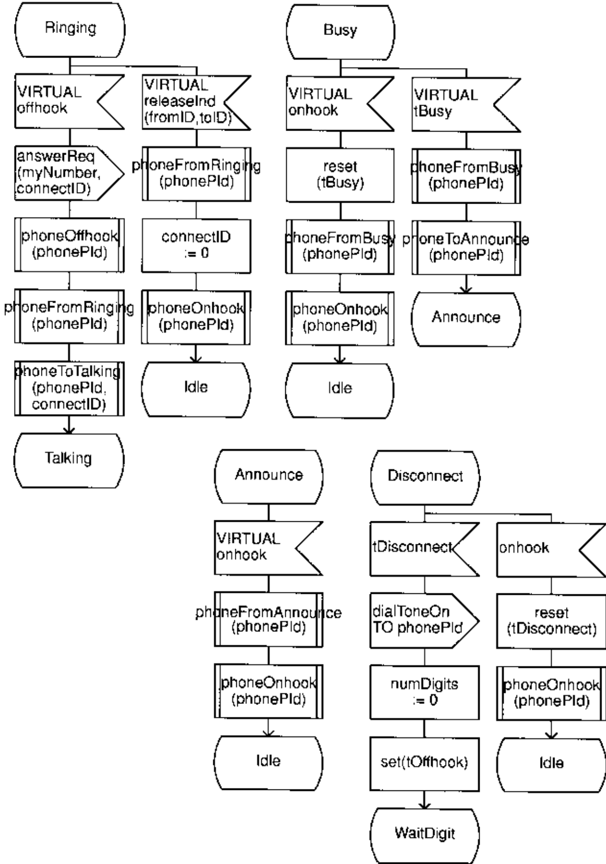


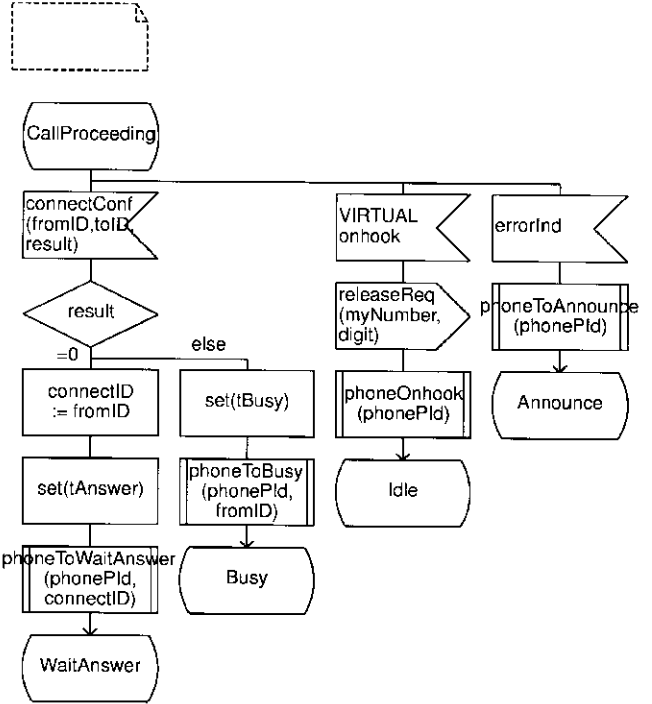
Appendix B. SDL model

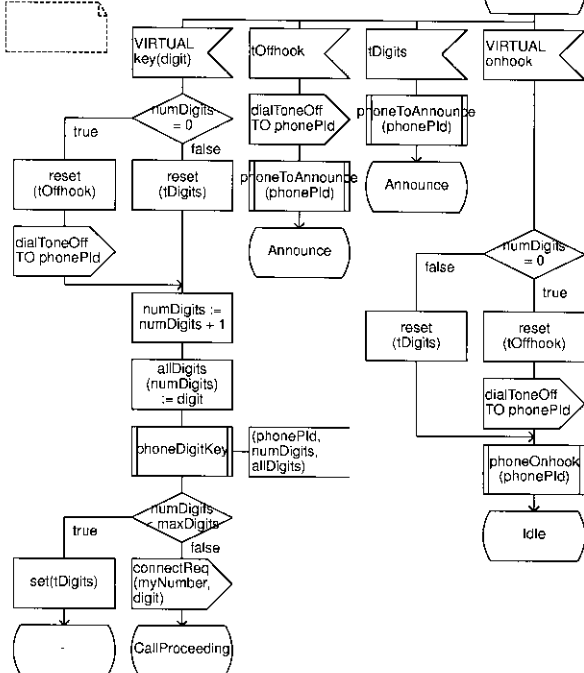
Here is our SDL model for the call processing part of one side of a basic phone call, and with various display features on the phone sets.

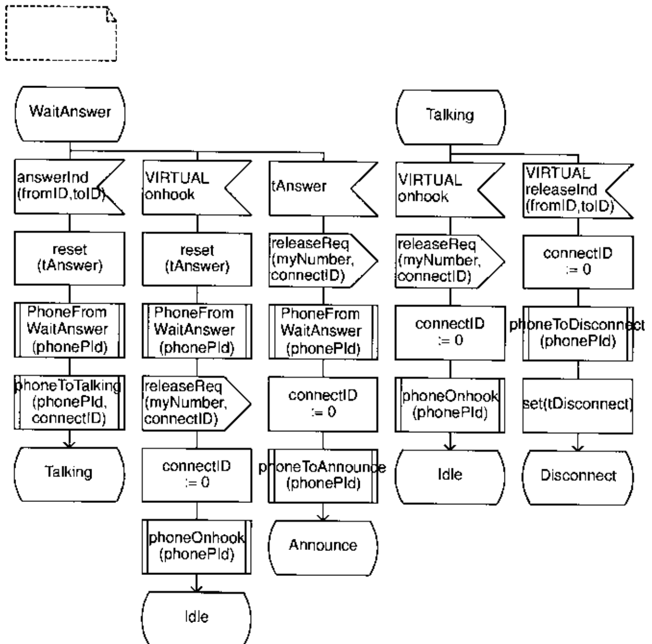
The procedure calls on the transitions can be swapped to change from a basic phone with no features, to one with several display lamps and a screen.




Virtual Process Type
<<System Type 4001System/Block Type Switch>>
CallProcessorType
Ringing
VIRTUAL offhook
VIRTUAL releaseInd (fromID,toID)
answerReq (myNumber, connectID)
phoneFromRinging (phonePId)
phoneOffhook (phonePId)
connectID := 0
phoneFromRinging (phonePId)
phoneOnhook (phonePId)
phoneToTalking (phonePId, connectID)
Idle
Talking
Busy
VIRTUAL onhook
VIRTUAL tBusy
reset (tBusy)
phoneFromBusy (phonePId)
phoneFromBusy (phonePId)
phoneToAnnounce (phonePId)
phoneOnhook (phonePId)
Announce
Idle
Announce
VIRTUAL onhook
phoneFromAnnounce (phonePId)
phoneOnhook (phonePId)
Idle
Disconnect
tDisconnect
onhook
dialToneOn TO phonePId
reset (tDisconnect)
numDigits := 0
phoneOnhook (phonePId)
set(tOffhook)
Idle
WaitDigit

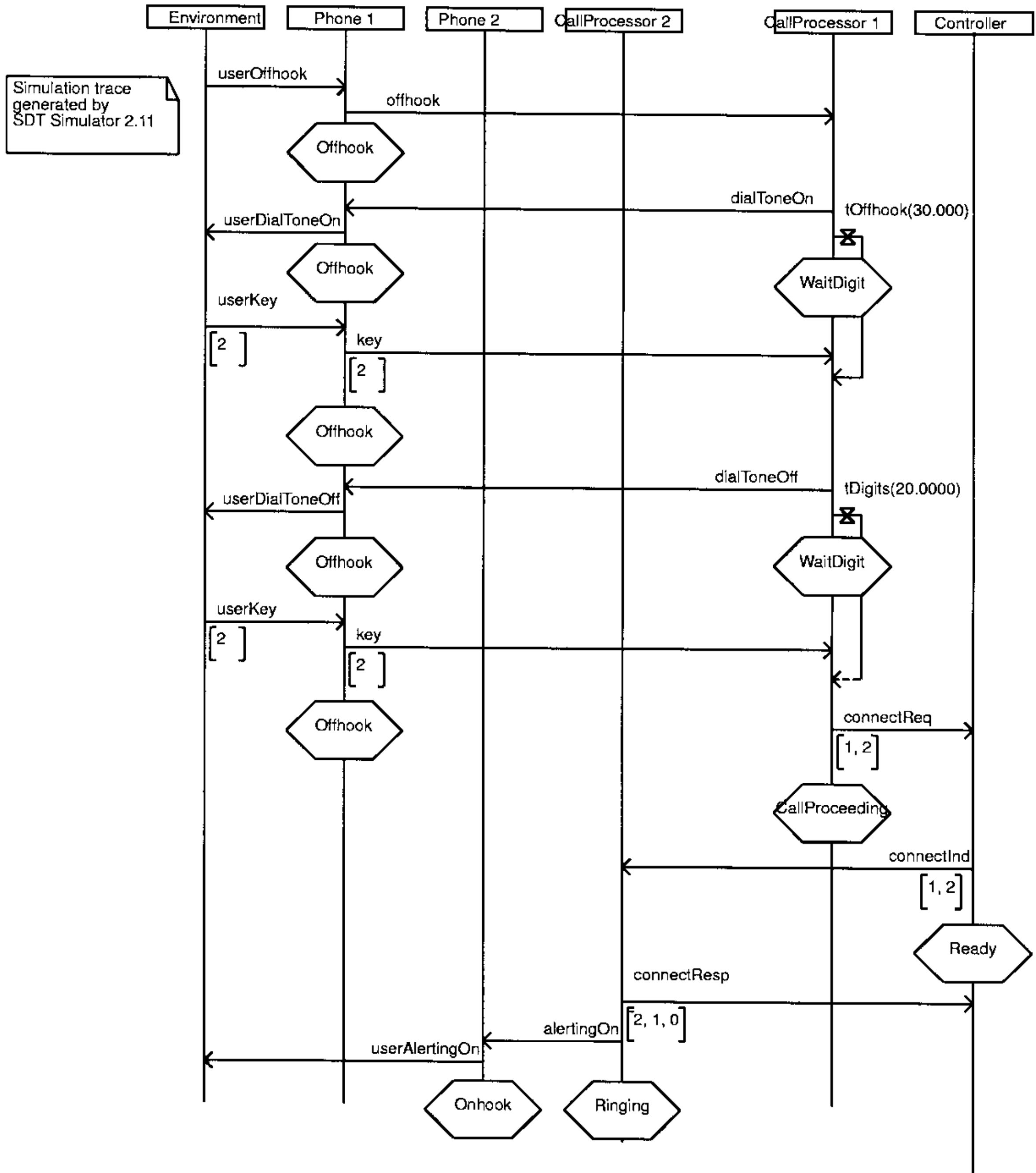

Virtual Process Type
<<System Type 4001System/Block Type Switch>>
CallProcessorType
CallProceeding
connectConf (fromID,toID, result)
VIRTUAL onhook
errorInd
result
=0
else
connectID := fromID
set(tBusy)
releaseReq (myNumber, digit)
phoneToAnnounce (phonePId)
set(tAnswer)
phoneToBusy (phonePId, fromID)
phoneOnhook (phonePId)
Announce
phoneToWaitAnswer (phonePId, connectID)
Busy
Idle
WaitAnswer

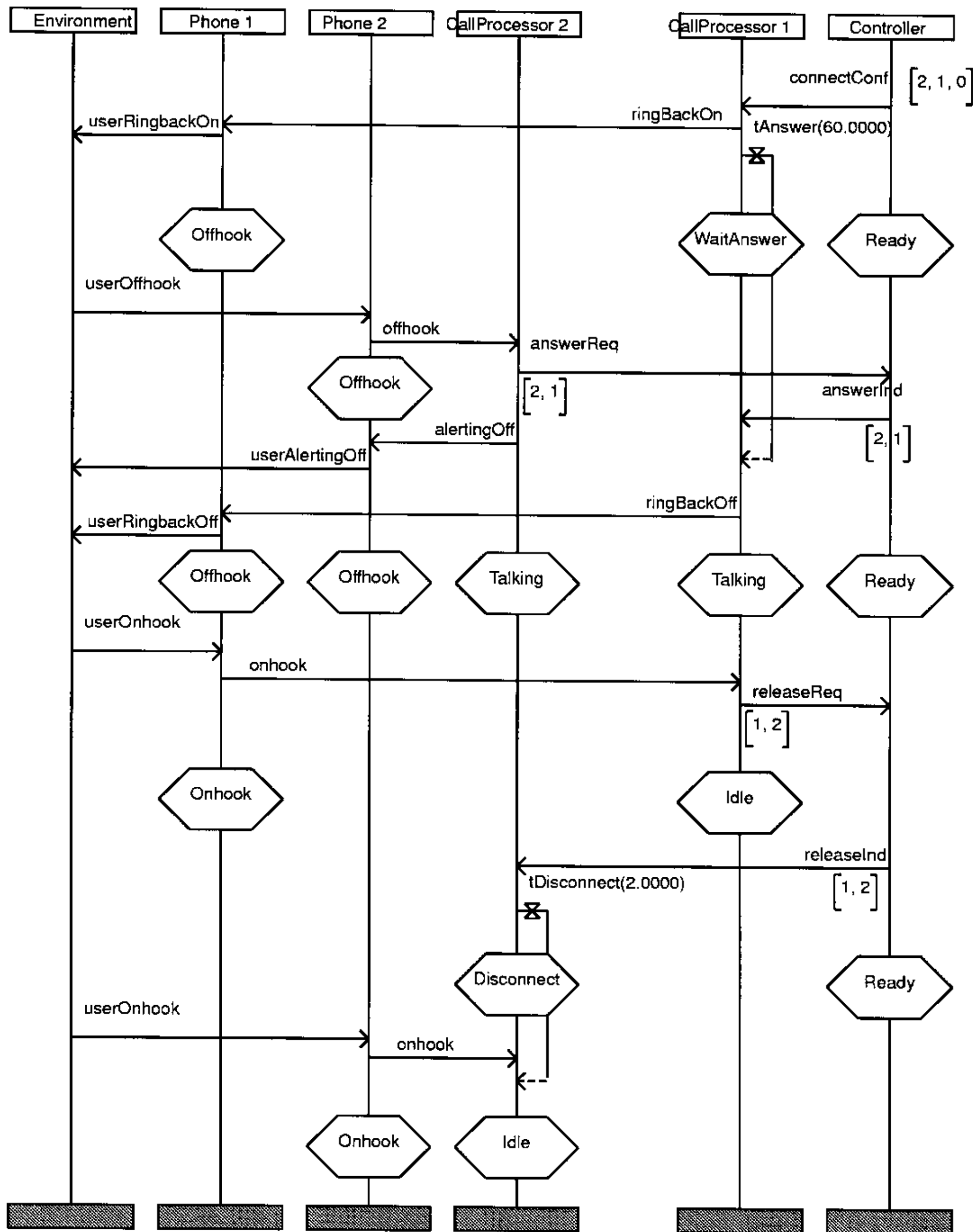

Virtual Process Type
<<System Type 4001System/Block Type Switch>>
CallProcessorType
WaitDigit
VIRTUAL key(digit)
tOffhook
tDigits
VIRTUAL onhook
numDigits = 0
true
false
dialToneOff TO phonePId
phoneToAnnounce (phonePId)
reset (tOffhook)
reset (tDigits)
phoneToAnnounce (phonePId)
Announce
dialToneOff TO phonePId
Announce
numDigits = 0
false
true
numDigits := numDigits + 1
reset (tDigits)
reset (tOffhook)
allDigits (numDigits) := digit
dialToneOff TO phonePId
phoneDigitKey
(phonePId, numDigits, allDigits)
phoneOnhook (phonePId)
numDigits < maxDigits
true
false
Idle
set(tDigits)
connectReq (myNumber, digit)
-
CallProceeding


Virtual Process Type
<<System Type 4001System/Block Type Switch>>
CallProcessorType
WaitAnswer
Talking
answerInd (fromID,toID)
VIRTUAL onhook
tAnswer
VIRTUAL onhook
VIRTUAL releaseInd (fromID,toID)
reset (tAnswer)
reset (tAnswer)
releaseReq (myNumber, connectID)
releaseReq (myNumber, connectID)
connectID := 0
PhoneFrom WaitAnswer (phonePId)
PhoneFrom WaitAnswer (phonePId)
PhoneFrom WaitAnswer (phonePId)
connectID := 0
phoneToDisconnect (phonePId)
phoneToTalking (phonePId, connectID)
releaseReq (myNumber, connectID)
connectID := 0
phoneOnhook (phonePId)
set(tDisconnect)
Talking
connectID := 0
phoneToAnnounce (phonePId)
Idle
Disconnect
phoneOnhook (phonePId)
Announce
Idle

Appendix C. Simulation MSC

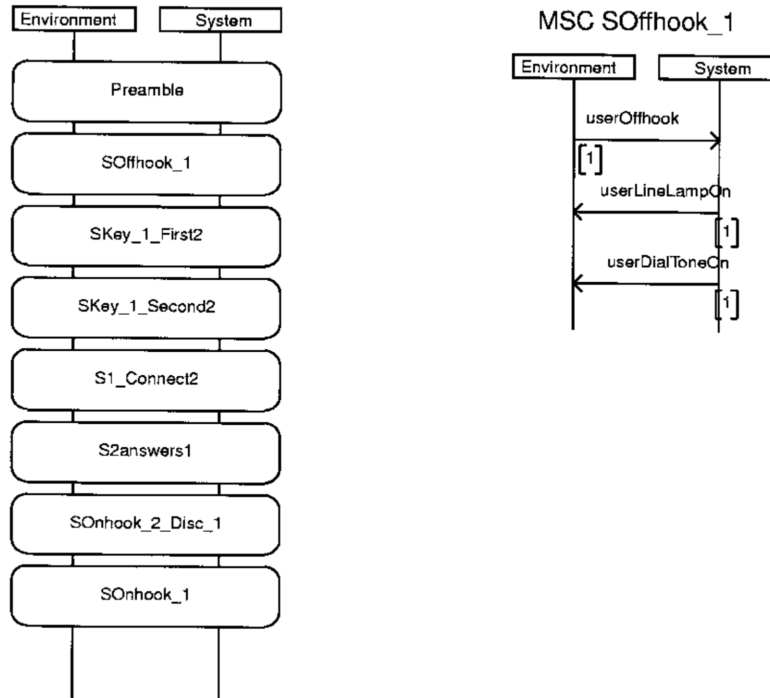
The following MSC is an example of one produced by simulated execution of an SDL model. It includes system substructure, timers, and SDL states.




Environment
Phone 1
Phone 2
CallProcessor 2
CallProcessor 1
Controller
connectConf
[2, 1, 0]
ringBackOn
userRingbackOn
tAnswer(60.0000)
Offhook
WaitAnswer
Ready
userOffhook
offhook
answerReq
Offhook
[2, 1]
answerInd
alertingOff
userAlertingOff
[2, 1]
ringBackOff
userRingbackOff
Offhook
Offhook
Talking
Talking
Ready
userOnhook
onhook
releaseReq
[1, 2]
Onhook
Idle
releaseInd
tDisconnect(2.0000)
[1, 2]
Disconnect
Ready
userOnhook
onhook
Onhook
Idle

Appendix D. MSC for test generation, including steps

The following shows an MSC composed of sub-MSCs. An example of one of the sub-MSCs is shown on the right. The sub-MSCs are used to indicate to the Autolink test generator that a sub-MSC will form a TTCN test step.



Appendix E. TTCN test case, including test steps

The following is a TTCN test case generated by the MSC in Appendix D.

Test Case Dynamic Behaviour					
Test Case Name : normalCall					
Group :					
Purpose :					
Configuration :					
Default : OtherwiseFail					
Comments :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		+Preamble			
2		+SOffhook_1_1			
3		+SKey_1_First2_1			
4		+SKey_1_Second2			
5		+S1_Connect2_1			
6		+S2_Answer1			
7		+SOnhook_2_Disc_1_1			
8		+SOnhook_1			
Detailed Comments :					

The following is one of test steps contained above. The test generator creates this table based on the sub-MSC in Appendix D.

Test Step Dynamic Behaviour					
Test Step Name : SOffhook_1_1					
Group :					
Objective :					
Default : OtherwiseFail					
Comments :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		user ! userOffhook	offhook_1		
2		user ? userLineLampOn	lineLampOn_1		
3		user ? userDialToneOn	dialToneOn_1		
Detailed Comments :					

References

[1] International Telecommunications Union (ITU-T), Recommendation Z.120: Message Sequence Charts, revised 1996.

[2] International Telecommunications Union (ITU-T), Recommendation Z.100: Specification and Description Language, revised 1996.

[3] International Standards Organization (ISO), OSI Conformance Testing Methodology and Framework—Part 3: The Tree and Tabular Combined Notation, International Standard 9646-3, 1992.

[4] Telelogic Tau tool set, version 3.3, Telelogic AB, Malmo Sweden.

[5] L. Doldi, V. Encontre, J.-C. Fernandez, T. Jéron, S. Le Bricquer, N. Texier, M. Phalippou, Assessment of automatic generation methods of conformance test suites in an industrial context, Proceedings of the Ninth International Workshop on the Testing of Communicating Systems (IWTCS '96), Darmstadt Germany, Chapman and Hall, London, 1996, pp. 347–361.

[6] A. Ek, J. Grabowski, D. Hogrefe, R. Jerome, B. Koch, M. Schmitt, Towards the industrial use of validation techniques and automatic test generation methods for SDL specifications, Proceedings of the Eighth SDL Forum (SDL '97), Evry France, Elsevier, 1997, pp. 245–259.

[7] J. Grabowski, R. Scheurer, Z.R. Dai, D. Hogrefe, Applying SaMsTaG to the B-ISDN protocol SSCOP, Proceedings of the Tenth International Workshop on the Testing of Communicating Systems (IWTCS '97), Cheju Island Korea, Chapman and Hall, London, 1997, pp. 397–415.

[8] M. Schmitt, A. Ek, B. Koch, J. Grabowski, D. Hogrefe, Autolink—putting formal test methods into practice, Proceedings of the 11th International Workshop on the Testing of Communicating Systems (IWTCS '98), Tomsk Russia, Chapman and Hall, London, 1998, pp. 227–243.

[9] A. Kerbrat, T. Jéron, R. Groz, Automated test generation from SDL specifications, Proceedings of the Ninth SDL Forum (SDL '99), Montréal Canada, published by Elsevier, 1999, pp. 135–151.

[10] H. Kahlouche, C. Viho, M. Zendri, An industrial experiment in automatic generation of executable test suites for a cache coherency protocol, Proceedings of the 11th International Workshop on the Testing of Communicating Systems (IWTCS '98), Tomsk Russia, Chapman and Hall, London, 1998, pp. 211–226.

[11] O. Monkewich, SDL-based Specification and testing strategy for communication network protocols, Proceedings of the Ninth SDL Forum (SDL '99), Montréal Canada, Elsevier, 1999, pp. 123–134.

[12] A. Cavalli, B. Lee, T. Macavei, Test generation for the SSCOP-ATM networks protocol, Proceedings of the Eighth SDL Forum (SDL '97), Evry France, Elsevier, 1997, pp. 277–288.

[13] E. Pérez, E. Algaba, M. Monedero, A pragmatic approach to test generation, Proceedings of the Tenth International Workshop on the Testing of Communicating Systems (IWTCS '97), Cheju Island Korea, Chapman and Hall, London, 1997, pp. 365–380.

[14] L. Feijs, M. Jumelet, A rigorous and practical approach to service testing, Proceedings of the Ninth International Workshop on the Testing of Communicating Systems (IWTCS '96), Darmstadt Germany, published by Chapman and Hall, London, 1996, pp. 175–190.

[15] S. Mauw, G. Velting, Algebraic Specification of Communications Protocols, Cambridge University Press, New York, 1993.

[16] M. Anlauf, Programming service tests with TTCN, Proceedings of the 11th International Workshop on the Testing of Communicating Systems (IWTCS '98), Tomsk Russia, Chapman and Hall, London, 1998, pp. 263–278.

[17] N. Fischbeck, Experiences with ISDN Validation Models in SDL and Proposal for new SDL Features, Proceedings of the Eighth SDL Forum (SDL '97), Evry France, Elsevier, 1997, pp. 135–150.

[18] K. Karoui, R. Dssouli, N. Yevtushenko, Design for testability of communication protocols based on SDL specification, Proceedings of the Eighth SDL Forum (SDL '97), Evry France, Elsevier, 1997, pp. 151–164.

[19] R. Lai, How could research on testing of communicating systems become more industrially relevant? Proceedings of the Ninth International Workshop on the Testing of Communicating Systems (IWTCS '96), Darmstadt Germany, published by Chapman and Hall, London, 1996, pp. 3–13.

[20] C.H. West, Protocol validation—principles and applications, Computer Networks and ISDN Systems 24 (1992) 219–242.

[21] G.J. Holzmann, Design and Validation of Computer Protocols, Prentice-Hall, Englewood Cliffs, NJ, 1991.