Test generation from SDL-specifications using partial-order reduction methods

Roorda, Sjoerd

Award date:
1993

EB 467
7068

Author: S.J. Roorda
Professor: Prof. dr ir C.J. Koomen
Supervisor: ir E. Kwast

Date: September 1993

Vakgroep EB

Test generation from SDL-specifications using partial-order reduction methods

Sjoerd Roorda

ptt research

SDL
_ fact _

Preface

All good things come to an end. And even though I have feared sometimes, this one would not, my graduation work has reached its completion too. Not that there is no work left to do, on the contrary. I started out with one question and ended up with many more.

I have worked at PTT Research, the department of computer science in Leidschendam in the research project "SDL-fact". It was also my graduation project for the Eindhoven University of Technology, department of electrical engineering. My graduation professor, prof. dr. ir. C.J. Koomen was kind enough to "let me go" and gave me the chance to carry out my graduation project at PTT Research. My supervisor (and colleague) at PTT Research was ir. E. Kwast.

As much as I am proud to present this report to you, as the master-piece that ends my time as a student and is the start of the rest that is yet to come, I am sad that I have to leave. I enjoyed my time at PTT research very much and I would like to thank all people that made that possible. Thanks to Erik, who spent a lot of time helping me out and pointing me to the right direction. Also a big "thank you!" to Stan, Hans, Han, Hans, Ed, Jan, Willem and Iko for their support.

When graduating at PTT Research, you are not alone. I would like to thank all fellow students that I have met during the last 10 months. Things would definitely have been different they all would not have been there... Finally, I probably would not even have started this without my parents help and advise. I definitely would not have finished it without Annemiek.

Sjoerd Roorda
September 1993

But now, what happens now?
Well... that's for you to invent.
-Till the end of the world-

Summary

This report describes the work that has been done in de project SDL-fact. The work has been carried out from December 1992 to September 1993 at PTT Research in Leidschendam, department of computer science (INF). Goal of the project was to evaluate partial-order based reductions methods for the reduction of the state space explosion in the context automatic test generation methods. Much of the work that has been done at PTT Research on the subject of protocol conformance testing is bundled in toolset called the **PTT Conformance Kit**. The test generation methods that have been implemented in the Conformance Kit have been the starting point for this report. However, in contrast to the toolset, the means of specification of protocols that is used is not the finite state machine, but the formal description language ***SDL***.

First, an overview is presented of the existing techniques for automatic tests generation from ***SDL***-specifications. Problems that arise when using ***SDL*** as a specification language are:

- the limited observability and controllability in the implementation under test;
- the nondeterminism in the system;
- the state space explosion that occurs when creating an "internal representation" of the system.

The explosion of the number of states limits the applicability of current test generation methods to medium-sized protocols. The construction of an internal representation of the composite system is necessary for all test generation methods that are presented. This internal representation holds the possible states in which the system may reside and the possible state transitions. A similar representation must be built in order to perform a verification of the system. Therefore, reductions methods that have been developed for protocol verification are likely to be applicable to the generation of conformance tests.

In the report, a number of heuristics-based methods are presented for the reduction of the state space representation of a system. The use of heuristics does not guarantee that the complete behaviour of the system still is described by the reduced system. Holzmann et al. have presented new reduction methods based on partial orders for protocol verification. These methods can be used to reduce the state space while preserving completeness of the model. The idea behind the partial order based reduction methods is that in concurrent systems not all actions are ordered (i.e. it is not defined whether an action should be executed *before* or *after* an other action). This is expressed in a "normal" representation by the presence of both interleavings of the actions. A reduction can be achieved by allowing only one interleaving of the un-ordered actions in the representation. The algorithms that accomplish this reduction for the protocol verification problem are:

- reduced search method;
- sleep set method;
- stubborn set method.

The algorithms that implement these methods block state transitions in the state space such that not all interleavings of un-ordered actions are present in the state space. The problem when these methods are applied to the generation of conformance tests is that the implementation, due to nondeterministic behaviour, may still select any of the blocked transitions. The outline for a new test generation method based on ***SDL*** is presented. In this method, the tester does not expect sequences of messages but uses sets of messages (called output sets) that must be observed. For each possible set, a (partial) ordering is defined.

It is likely that this new method can be used successfully for the reduction of the state space. Although the reduction that can be achieved will be much smaller than in the case of protocol verification, improvements can be achieved in:

- the use of resources (such as CPU-time and memory);
- the amount of test that is generated;
- the fault coverage of the generated test.

Open questions are the relation between the test method based on output sets and the theory that has been developed in the project ARTEFACT. Also, the effects of the assumption of a "reasonable environment" on the fault coverage of the tests in not clear.

Contents

1 Introduction **1**

2 Conformance testing **5**

2.1 Specifications 5

2.1.1 Formal description techniques 6

2.1.2 Incompleteness of specifications 7

2.2 Conformance testing methodology 8

2.2.1 Abstract test case definition 10

2.2.2 Architecture 11

2.2.3 Modeling the state space 14

2.2.4 Representations 17

2.2.5 Completion of the specification 18

2.2.6 Conformance relations 19

2.2.7 Testing strategy 21

2.3 Composition of finite systems 22

2.4 Testability 24

2.4.1 Nondeterminism 24

2.4.2 State space explosions 24

2.4.3 Fault coverage 25

2.4.4 Design for testability 25

3 Automatic test generation **27**

3.1 Introduction 28

3.2 FSM-based test generation techniques 29

3.3 Tree-based test generation techniques 32
3.4 Test generation based on canonical testers 32
3.5 PTT Conformance Kit 33

4 SDL specifications **35**
4.1 Structure of SDL specifications 36
4.2 Observable behaviour 37
4.3 Nondeterminism 38
4.4 Modeling the state space of SDL-specified systems 40
4.4.1 Finite state machines 41
4.4.2 Labeled transition system 41
4.4.3 Asynchronous communication trees 43
4.4.4 SAVE constructs 44
4.5 Simple handshake example 45
4.5.1 Description of the example 45
4.5.2 Nondeterminism in the example 46
4.5.3 Infinity of the state space 46
4.6 Testing SDL-specified systems 47
4.6.1 ACT-based test methods 48
4.6.2 SDL-machines 49
4.6.3 Behavioral constraints 50

5 Partial order reduction strategies in protocol verification **53**
5.1 Protocol verification 53
5.1.1 Reachability analysis 55
5.1.2 State space explosion 56
5.2 Reduction methods using heuristics 56
5.2.1 Improving the specification 57
5.2.2 Splitting the specification 57
5.2.3 Partial searches 57
5.3 Reduction methods using partial orders 59
5.3.1 Reduced search method 59

5.3.2 Sleep Sets 61
5.3.3 Stubborn Sets 63

6 Application of the partial order reduction methods 65
6.1 Controllability of traces 65
6.1.1 The LTS model 66
6.1.2 fairness 66
6.1.3 controllability 67
6.2 Stable traces 68
6.2.1 Non-stable states 68
6.2.2 Existence of stable traces 72
6.2.3 Projection sets 73
6.2.4 Controllable behaviour 74
6.3 Generating tests for SDL-specified systems 76
6.3.1 Output sets 76
6.3.2 Observation of the output sets 78
6.3.3 Refinement of the output sets 79
6.3.4 Notation 80
6.4 Application of the reduction methods 81
6.4.1 Adoption of the algorithms 83
6.4.2 Test procedure 84
6.4.3 Fault coverage 84

7 Conclusions 87

A The reduced search method 95

B Conflict set method 99

C Stubborn sets 106

D Two independent processes 112

E Examples stable traces 117

1 Introduction

Telecommunications is governed by protocols. The use of telecommunications extends ever further by integrating different media of communications like speech, data, pictures and video and by accomplishing higher speeds and better availability. The protocols that accomplish this grow increasingly more complex. In a modern telecommunications architecture, a large number of protocols are used at the same time so the environment for these protocols also becomes more complex. The complexity of modern protocols poses some important problems. Means are needed to help designers and users of telecommunication protocols to deal with this complexity.

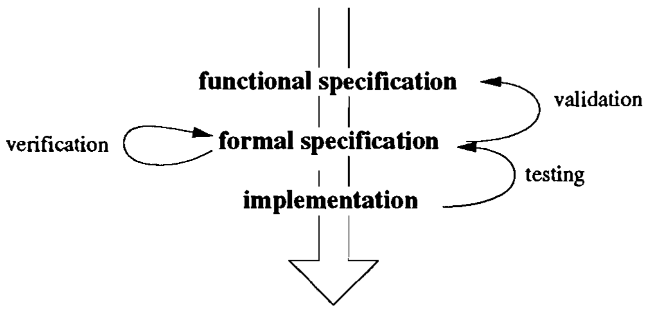


Figure 1.1: Basic design cycle for a protocol

A typical design process for a telecommunications protocol (the **design cycle**) is given in figure 1.1 [Koo91]. Three phases can be distinguished in the design process. A **functional specification** is made in the first phase. This functional specification defines *what* the protocol should do. The **formal specification** which is constructed during the second phase precisely defines *how* the protocol should accomplish these tasks. Finally, an **implementation** is built according to the formal specification that implements the wanted functionality. Very important in the design process is the recursion to a previous phase to check whether the current steps in the process comply with the previous phases (figure 1.1).

In the design cycle there are different aspects that have become increasingly important with the growing complexity of protocols. **Formal description techniques** (***FDT***'s) are becoming very important in the second and third phase of the design. The techniques allow the designer to deal with complexity by using mathematics and computers. Also, various other parts of the design cycle can be automated when ***FDT***'s are used. An other essential aspect for the correct design of a complex protocol is extensive **verification** and **testing**. Only through these techniques the designer is able to ensure he ends up with a protocol that actually does what it should do. In this report, testing is defined as checking whether the implementation correctly implements the specification, while verification checks for logical errors in the specification. The automation of testing (especially the generation of tests) and verification with the help of formal description techniques has been an important issue during the last years.

The automated generation of tests in order to test a protocol to its specification is the main subject of this report. At PTT Research, much work has been done on the field of automatic test generation from protocol specifications. Much of the results of this work has been bundled in a toolset called "**PTT Conformance Kit**" [BKKW91]. This toolset implements different test-generation techniques based on specifications that are written down as **finite state machines** (***FSM***'s). The Conformance Kit has been used to generate tests for practical protocols. All test-generation techniques, including the ones that were implemented in the Conformance Kit, are limited to medium-sized protocols. To keep up with the complexity of modern protocols, it is necessary to improve the existing test generation methods such that also tests for very large protocols can be generated automatically.

This report describes the project **SDL-fact**. This project is a continuation of some large research projects at PTT Research. The goal of the project was to examine some recently presented reduction techniques from the field of protocol verification in the context of the test generation methods that were developed for the PTT Conformance Kit such that tests can be generated for larger protocols. With these reduction techniques, that were presented by G. Holzmann et al. [HGP92], it should be possible to apply existing test generation techniques to very large protocols. One of the questions that had to be solved in the project was: "Can these reduction techniques be used in cooperation with the test generation techniques that have been implemented in the PTT Conformance Kit?" Also, one important demand was added: the description technique that was to be used, was not the finite state machine (which is used in the Conformance Kit) but the formal description language ***SDL*** (Functional Specification and Description Language).

The report is set-up as follows: First a general overview is given of existing testing techniques. Some possible architectures are considered and the problem of state space explosions is introduced. Secondly, some automatic test generation methods are considered. After this, the attention is shifted to the specification language ***SDL***; some basic concepts of the language are presented and specific problems related to automatic test generation from ***SDL***-specifications are summarized. The last part of the report presents some approaches in reducing the representation of a protocol that are known in the field of protocol verification (including the reduction methods by Holzmann et al.). The application of these methods to the problem of generating tests from ***SDL***-specifications is covered in the last part of the report.

2 Conformance testing

Testing whether the observable behaviour of an implementation conforms to the relevant specification is called **conformance testing.** In this chapter, a number of general aspects of conformance testing are summarized. The arguments that are given here are applicable to both manually and automatically generated tests.

In order to obtain some degree of certainty about the correct operation of a protocol implementation, a very large number of test is needed. With the help of standardized test description languages these tests can be executed automatically. Still, the reader should bear in mind that there is a principal difference between a specification of a protocol (which is an abstract representation of the systems behaviour) and an implementation of a protocol (a physical system). Tests that are generated for physical systems must be executed in a finite amount of time and with a finite amount of resources[1].

2.1 Specifications

One of the most important reasons for the rapid progress in telecommunications today is the availability and widespread use of standardized protocols. Making the standards is a slow and awesome process, yet the benefits are numerous. The most important institutions that contribute in the development of standardized protocols are the CCITT, ISO and ETSI. The standardization of communication protocols allows consumers to choose freely between products of different manufactures. With these products, large multi-vendor telecommunication architectures can be built that meet the demands of modern telecommunications.

When interconnecting different building blocks of a telecommunication architecture such as or protocol layers or complete protocol implementations, it is very important that each separate building block conforms to its specification (e.g. that it does what the specification prescribes). This is especially important when the user wants to inter-

[1]To make a set of generated tests "usable" in a practical system, the finiteness is not sufficient. In this case practical and economical arguments are even more important

connect protocol implementations made by different manufacturers. Even though each implementation of a protocol is built according to the same standardized specification, this does not guarantee correct cooperation of different implementations. A number of reasons can be found for this problem:

- implementation errors;
- different interpretations of the specification;
- incompleteness of specifications;
- implementation of different options.

Due to errors during the design process, an implementation may not fully comply to the specification. Often, these errors only show up in very complex situations or even only when the implementation is connected to products of different manufacturers. Other causes of possible deviation from the standard are considered in the following paragraphs.

2.1.1 Formal description techniques

A very important problem for the creators of protocol standards is to write down the definition of the behaviour of the protocol unambiguously. It has appeared that this is a very difficult task. Almost all standards are written in English, which makes it almost impossible to exclude ambiguities in the specification. Formally, protocols that are built according to different interpretations of the same specifications all conform to the specification. However, such situations must obviously be avoided.

An important solution for the problem of ambiguities in specifications is the growing use of formal specification languages (see table on page 7). These languages can be used to describe the behaviour of a protocol. The semantics of these languages allow only one interpretation. The formal description of protocols also allows for the specifications to be processed by computers. Large parts of the design cycle of a protocol can be (partially[2]) automated this way. This report focusses on automatic verification of protocols and automatic test generation for protocols. Besides those applications, also automatic implementation could be considered when using formal languages to describe protocols.

The introduction of formal description techniques in the whole protocol design process is a difficult task. The choice between the available formal languages is not obvious and depends on a number of factors such as the kind of protocol, the availability of tools and

[2]Most approaches toward automating test generation assume that some parts in the design process are still accomplished manually. Typically, the automation of parts of the design process serves as an aid to the designer.

Standardized formal specification languages

i **Functional Specification and Description Language (*SDL*)**:
The language started out in 1976 merely as a collection of standardized pictures that could be used to describe systems. It has been recommended in different versions: SDL80, SDL84, SDL88 and recently the latest version: SDL92 was accepted by the CCITT. In this report, the 1988 version of ***SDL*** is used, unless stated otherwise. The language has a graphical and a phrase representation. It is based on extended finite state machines. Much effort is done to promote ***SDL*** as a means of specifying protocols [IC92].

ii **LOTOS**:
The formal language LOTOS was developed for scientific purposes. It has been used in the development of protocol testing and verification techniques and is based on the temporal ordering of observational behaviour. It is standardized in the ISO 8807 international standard [ISO88].

iii **Estelle**:
The structure of Estelle is very much alike ***SDL***. The language was standardized by the ISO in the ISO 9074 international standard [ISO89]. Estelle was developed as a means of removing ambiguities in the specifications of layers of the Open Systems Interconnection (***OSI***) architecture, defined by ISO.

Figure 2.1: Formal specification languages.

fashion. Each language has its own strengths and weaknesses which make it more or less suitable for specifying certain kinds of behaviours and allow for easier application of some automated design techniques. However, with the growing availability of tools that support formal languages, the formal description techniques become more and more important.

Often, it still is very hard to automate the generation of tests from the specifications in formal languages. Later in this report some techniques will be presented that translate the high-level formal languages to lower-level models of the protocol that can be used for automatic test generation or automatic verification like finite state machines and labeled transition systems.

2.1.2 Incompleteness of specifications

Often specifications of protocols are not complete [BEVT89]. However two kinds of incompleteness can be distinguished. The first is incompleteness that results from the fact that parts of the protocol are not or only partially specified. The exact completion is left to the manufacturer. This will inevitably lead to different implementations of the same specification. The formal methods that were described in the previous paragraph can be used to determine whether a specification is complete or not.

A second kind of incompleteness results from implementation options that are offered to the manufacturer. Here all possible specifications are well defined. In this case it is important that the manufacturer specifies exactly which options were chosen. This procedure is standardized and described in the ISO 9646 standard called "Conformance Testing Methodology and Framework" [ISO90]. A document describing the chosen options is called a ***PICS*** (*protocol implementation conformance statement*). A second document gives extra information about the implementation that is necessary to be able to test it to its specification. Such a document is called a ***PIXIT*** (*protocol implementation extra information for testing*).

In the rest of this text it will be assumed that a complete formal specification is available, completed with the ***PICS*** and ***PIXIT***-documents. Further, it is assumed that the specification is *errorless* (i.e. that a complete and rigorous verification of the specification is detected).

The availability of complete standardized specifications inevitably raises the question whether a certain implementation conforms to its specification. Testing whether an implementation of a protocol conforms to its specification is called **conformance testing.** It is important during the design process of the protocol implementation to know whether the implementation still conforms [Hol92]. Therefore, conformance testing is an important part of the design process that is done by the manufacturer. On the other hand, users of a specific product want to conduct there own conformance tests to ensure correct implementation of a standard and interoperability with its existing products.

2.2 Conformance testing methodology

With the growing importance of conformance testing for all parties in the telecommunication field, the need emerged for a standardized approach towards conformance testing. The most important contribution in this field is the ISO 9646 international standard. Their effort is motivated by the following statement (OSI stands for a large range of standards and recommendations made by ISO):

> The objective of OSI will not be completely achieved until systems can be tested to determine whether they conform to the relevant protocol standard(s) or recommendation(s). [...] Standard test suites should be developed for each OSI protocol standard, fur use by suppliers or implementors in self-testing, by users of OSI products [...] or by other third party testing organizations. This should lead to comparability and wide acceptance of

test results produced by different testers, and thereby minimize the need for repeated conformance testing the same system.

Two different properties of an implementation under test (*IUT*) determine its conformance to the specification.

- static conformance requirements;
- dynamic conformance requirements.

The static conformance requirements define the legal combinations of options or parameters ranges. The dynamic conformance requirements define the *external* behaviour that is permitted by the relevant standards. In the remainder of this report the term "conformance testing" will be used to denote tests that only test the dynamic conformance requirements.

Two testing methodologies can be distinguished [Hol91a]. The first: *functional testing*, is a mere translation of the demands of the administrators of the first multi-party networks into testing objectives. This kind of testing could work to a reasonable extend for small systems but is not suited for larger applications. *Structural testing*, is a more structured approach that evolved from functional testing to deal with the growing complexity of the protocols.

- **Functional testing:** establish that a given implementation realizes all functions of the original specification, over a full range of parameter values and reject erroneous inputs in a way that is consistent with the original specification.
- **Structural testing:** establish that the control *structure* of the implementation conforms to the structure of the specification. Implementation and specification have the same control structure if they model *equivalent* sets of states and allow for the same state transitions.

From the definition of structural testing it follows that a model of the protocol is needed that inhibits the possible states of a protocol. Since the specification to which the instance is tested is only a model of the real protocol, it is important that the model is as close to the implementation as possible. Testing whether the implementation under test (*IUT*) inhibits the same control structure as the specification has great advantages. Structural testing is well suited for automated test generation. It is also easier to standardize structural tests as it is to standardize functional tests. Therefore, only structural testing will be considered in this report. Yet, a system failing a structural test could implement the functions correctly, only with a different control structure. Formal description techniques are becoming more important in writing specifications, which helps closing this gap.

It is important to notice that *control structure* and *internal structure* are two different properties of a protocol implementation. A structural test is based on a notion of equivalence between the control structures of the implementation and the specification. However, when conducting a conformance test, the protocol instance can only be tested as a black box. When testing a system as a black box the *internal* structure is assumed irrelevant [Hog91a]. Only the interaction with the environment plays a role (often, this is also the only part of the behaviour that was standardized). The flow of this interaction depends on the control structure of the protocol.

Communication protocols are designed to transport information. Very often the functionality of the protocol does not depend on the exact contents of the exchanged data. For instance, the words that are exchanged during a telephone call do not influence the operation of the telephone services. The first step in dealing with the complexity in protocols is abstracting the contents of the exchanged data from the model. In structural testing a rigorous abstraction will be made. Consequently, not all possible behaviour of the protocol is confined in the resulting model. After the generation of the tests, the designer of the tests will have to find relevant values for the data that was excluded from the model. Relevant values often are upper and lower limits that are defined for the system. A complete conformance test should also test whether the data is properly processed (i.e. whether the computations of the protocol are correct). This aspect, called **data testing** is not a part of this report.

2.2.1 Abstract test case definition

Both for the manufacturers and the customers it is very important that it can be established whether a protocol implementation conforms to a specification or not. Instead of testing each implementation to all others, only one test is necessary. However, it is necessarily that conformance tests are commonly available and accepted by all parties in the design process. Also the results of the conformance tests must be standardized and accepted. If this is accomplished, test results of manufacturers or independent test institutions can serve as a reference for all parties, eliminating the need for repeatedly testing of the same system. As mentioned, the ISO "Conformance testing Methodology and Framework" ([ISO90, part 1&2]) is an important document that is intended to standardize the way conformance tests are made and used. One part (part 2, "Abstract test suite specification"), defines a way of defining conformance tests, independently of the means of executing the tests. Figure 2.2.1 gives an overview of the terms associated with the abstract test suite specification.

The third part of the ISO 9646 standard deals with the definition of a test specification language called "Tree and Tabular Combined Notation" (***TTCN***). ***TTCN*** is commonly adapted by manufacturers and users as the language in which tests are specified. With this notation, the tests can be applied directly to the implementation (either manually or automatically). The tests that are generated by the PTT conformance Kit

ISO 9646, Abstract Test Suite Definition

In [ISO90, part 2], the concept of **abstract test suites** is defined. An abstract test suite can be divided into different abstraction levels. In the highest level, a specification of the tests is presented which is independent of the actual implementation of the protocol *and* the testing environment. On a lower level of abstraction, different **test groups** can be distinguished (e.g. conformance, performance). A group can be divided in multiple sub-groups (e.g. call-setup, data exchange and termination). At the lowest hierarchical level, the **test cases** are defined. Each test case consists of a **test purpose** and a **test sequence**. This test purpose defines exactly which part is tested and in which way. The test sequence is a sequence of inputs to the implementation and the expected outputs.

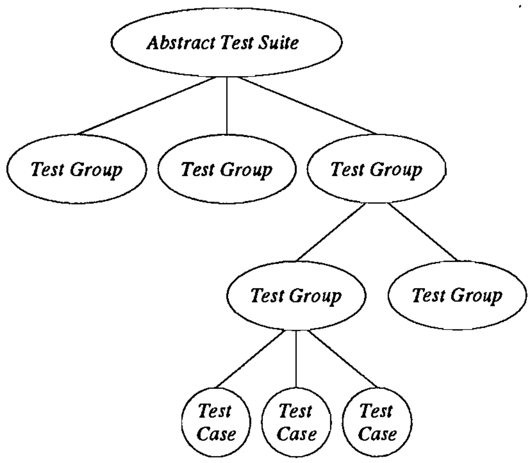


Figure 2.2: Abstract Test Suite definition.

are specified in ***TTCN*** and optionally also in a private dialect: RNL-***TTCN***. This dialect is more readable and can be compiled by interfaces to automatic testers. For the definition of the language see [ISO90, part 3].

2.2.2 Architecture

A conformance test is done on an existing product. Usually there is not much information available about the internal structure. Most protocol standards only define the observable behaviour of a protocol. For an important class of protocols, defined in the Open Systems Interface (***OSI***) architecture this is stated as:

Only the external behaviour of Open Systems is retained as the standard of behaviour of real Open Systems.

The tester can only access the interface between the *IUT* and the environment. This communication happens at a limited number of predefined points called **points of control and observation** (***PCO***'s). The manufacturer of the protocol could add extra ***PCO***'s, for instance to facilitate the testing. The tester is not able to see an other information than what appears at one of the ***PCO***'s. Conformance testing, therefore, is **black box** testing. So the verdict of a conformance test can only be that the external behaviour of a system conforms to the external behaviour that could be expected from a system with the same control structure[3]. A protocol exchanges messages, called ***PDU***'s[4] or ***ASP***'s[5] with its environment. In the ideal case, these messages can be exchanged with the *IUT* directly. In [P5493], a more general applicable architecture for testing a communication protocol is given. In figure 2.3 this architecture is drawn. The tester can not access the ***IUT*** directly but communicates at the ***PCO***'s via the test context. In [ISO90], the ideal case, in which the messages can be observed and controlled

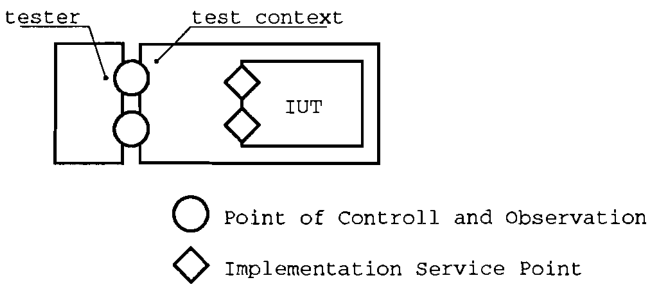

tester
test context
IUT
Point of Controll and Observation
Implementation Service Point

Figure 2.3: General testing architecture

directly at the boundaries of the protocol is defined (figure 2.4). The testing method related to this (abstract) architecture is called the **local test method**. In practical cases,

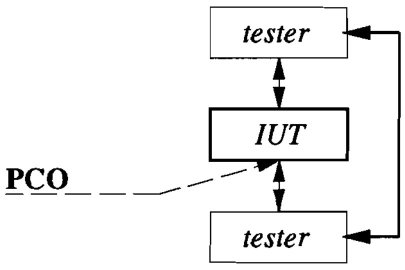


Figure 2.4: Local testing architecture

the tester will not be able to apply or observe the messages directly at the boundary

[3]The verdicts of a tester are denoted as ***pass***, ***fail*** or ***inconclusive***. The latter being passed in situations when the system is non-deterministic.

[4]PDU = Protocol Data Unit. PDU's are exchanged in a protocol layer.

[5]ASP = Abstract Service Primitive. An ASP is exchanged with higher or lower layers.

of the protocol. A situation like figure 2.5 is more realistic. In this situation (*remote testing*) there is only a *virtual channel* between the tester and the protocol instance. In the literature, the term "tester" is both used for the unit that executes the tests and for the person that is responsible for conducting the tests. In this report, the latter will be denoted as **test person** if it is not clear from the context. A tester is defined in definition 1.

Definition 1 (Tester) *A tester is a unit which applies the input portion of a test-sequence to the implementation, observes the outputs, and compares them with the expected outputs [SD88].*

Often the tester is split in multiple parts. Each part generating I/O sequences for a specific *boundary* of the protocol. In the local test method (figure 2.4), this would yield two testers: an **upper** tester that acts like an application on the the ***IUT*** and a **lower** tester that communicates with the bottom-side of the ***IUT***. These testers also have to inhibit functionality that enables them to transform the (abstract) test-suites to physical signals and visa versa. In a practical architecture like figure 2.5, upper and lower testers can also be distinguished. In this case, the lower tester can not communicate directly with the ***IUT***, but sends the ***ASP***'s via a *virtual channel*. In order to be able to perform a conformance test it has to be assumed that both the communication between the testers and the ***IUT*** and the translation of the signals is errorless.

The fact that only the external behaviour of the protocol implementation is accessible for the tester and that even this accessibility is done indirectly via other protocol layers limits the possibilities of testing in two ways. First the **controllability** is limited since the system can not directly be put in an arbitrary state. Specific sequences of inputs are necessary to bring the system to a certain state. Second, since the only information about the system's internal state can be obtained from its outputs, the **observability** is restricted to what can be deduced from the outputs [DSU90].

For systems that consist of large protocol stacks, this kind of architecture puts a severe limitation on the possibilities of testing the implementation since the messages

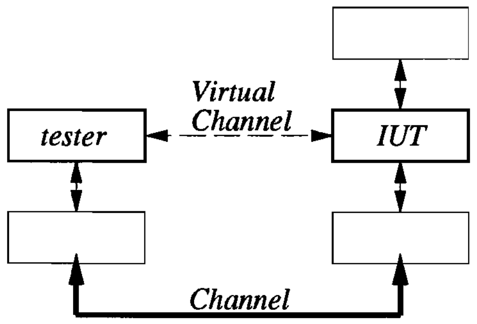


Figure 2.5: Remote testing architecture

have to pass through all layers of the stack before arriving at the ***IUT***. From a testing point of view it would be very desirable to be able to apply test sequences directly at the boundaries of the implementation. Techniques as *ferry-clip* and *boundary scan*[6] would give the tester a way to apply signals directly to the borders of the implementation. A lot of effort will be needed before those methods can be standardized.

2.2.3 Modeling the state space

Most existing test generation techniques assume a somewhat different model of the protocol than the formal description languages like ***SDL*** or LOTOS. Such a model is called an **internal representation** or an **intermediary model**. The information that is needed is:

- possible states in which the system may reside;
- possible state transitions;
- inputs and outputs associated with the possible state transitions.

The models that are described in this report are the **labeled transition system** (***LTS***), the **(extended) finite state machine** ((***E***)***FSM***) and the **behavioural tree**. The labeled transition system is the most versatile model. It hardly puts any restrictions on the protocol. This feature, however, makes it less suitable for the application of automatic test generation techniques. A short introduction for all models will be given here:

labeled transition systems

A labeled transition system consists of all possible states that can be reached by the system. States are different if the future behaviour of the system in that states may differ. A system may transfer from one state to an other by executing an **action**. This action is tagged as a label to the relevant transition (hence the name labeled transition system). Formally the ***LTS*** is described as a 3-tuple:

$$L = (S, A, \delta)$$

- S = set of states $\{s_1, s_2, \ldots\}$
- A = alphabet: finite set of actions $\{\alpha_1, \alpha_2, \ldots, \alpha_q\}$

[6]Boundary scan was introduced as a method in the design of integrated circuits to enable individual chips to be tested separately form their environment but after they were mounted on the board.

- δ = *partial* transfer function $\delta : S \times A \rightarrow S$

Often a special action, the τ-action is added which models some "internal" behaviour. Such an internal action is not observable or controllable from the outside of the system. Such a situation might occur when there is actually some behaviour that is not known. Often internal actions are used to mode behaviour that is excluded from the model in an abstraction step. In such cases, τ-actions may model a choice on the basis of information that is not in the model (figure 2.6a) or the manipulation of a variable that is abstracted from the model (figure 2.6b).

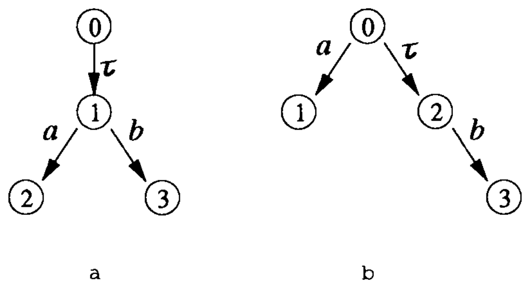


Figure 2.6: τ-actions in the labeled transition model

The use of internal action may introduce non-deterministic behaviour into the model (as in the figure). Note that nondeterminism may also be present in the model, even without internal actions. There are different ways to deal with non-determinism. Some methods that can be used to deal with nondeterminism will be described later.

finite state machines

Most test generation methods use a model based on finite state machines. This model resembles the labeled transition system. However, a number of restrictions is added. In this report a special kind of ***FSM*** is used: the Mealy Machine [fsm69]. For this ***FSM***, the state transitions are defined by input/output pairs. There is only a finite number of possible states. The system transfers from one state to a next with the consumption of an input and the output of the output message of the relevant input/output pair. The Finite State Machines that are used are described as a 5-tuple:

$$M = (S, I, O, \delta, \lambda)$$

- S = finite set of states $\{s_1, s_2, \ldots, s_n\}$
- I = finite set of inputs $\{i_1, i_2, \ldots, i_q\}$
- O = finite set of outputs $\{o_1, o_2, \ldots, o_r\}$

- δ = transfer function $\delta: S \times I \rightarrow S$
- λ = output function $\lambda: S \times I \rightarrow O$

In the following text the ***FSM***s that are used must obey some assumptions. There are also some methods for state machines that have different properties, but for reasons of simplicity, these are not presented here.

1. The number of possible states is finite.
2. The system that is modeled by the ***FSM*** is **deterministic**. This means that in a specific state, a specific input will always cause the same, single, transition.
3. From any state that is reachable by the ***FSM***, any other state can be reached via a finite sequence of transitions.
4. There is a known initial state: S_0.

extended finite state machines

A finite state machine can be extended to hold predicates and variables. Such a model is called an **extended finite state machine e*FSM***. The restrictions on the number of states and the nondeterminism are the same as in the case of a normal ***FSM***. However, if the variables in the e***FSM*** are not bounded (i.e. if the possible values of the variables are not countable or finite), the total number of possible states in which the system may reside is infinite.

In contrast with a normal ***FSM***, multiple transitions may be defined for a single input in a certain state. However, the system will chose one of the possible transitions on the basis of a predicate. A predicate is a boolean expression containing the available variables in the model.

An extended finite state machine can be translated to a normal finite state machine by assigning a unique state to each combination of variable values. The resulting number of states may become infinite. Often the number of states after the translation can be reduced drastically by determining which of the possible states are actually reachable by the system. An example part of an extended finite state specification, which is used in the PTT Conformance Kit is depicted in figure 2.7.

behavioural trees

A form that is very similar to the labeled transition model is the behavioural tree. This model was introduced by Milner [Mil80] as a means of modeling the state space of systems that were specified in ***CCS***. In general, a behavioural tree can be obtained

Example part of an extended finite state machine specification

```
EFSM connaisseur;

VARIABLES
        glasses: 0..5;                      -- tasted
START
        A: (glasses := 0);                  -- starting state
GATES
        glass, opinion, sound, question;
INPUTS
        bordeaux, AH_huisw, coffee @ glass;
        do_you_like_it @ question;
OUTPUTS
        bon, pas_bon @ opinion;
        klok_klok, klok_hips @ sound;
TRANSITIONS
-- predicate    input           output          action                  new_state
STATE A;
  (glasses<5)   bordeaux        klok_klok       (glasses:=glasses+1)    B;
  (glasses=5)   bordeaux        klok_hips       -                       B;
  (glasses<5)   AH_huisw        klok_klok       (glasses:=glasses+1)    C;
  (glasses=5)   AH_huisw        klok_hips       -                       B;
END

etc.
```

Figure 2.7: Example part of an extended finite state machine.

by unfolding a labeled transition graph. Such a tree represents all possible sequences of actions that may be executed by the system, starting form the starting (root-)state. The problem with a tree-model is the fact that it will, in general, not be finite for practical systems.

A derivate of the behavioural tree is the asynchronous communication tree (***ACT***), that was developed by Agha [Agh86]. An ***ACT*** models the observable behaviour of the system, and is therefore very well suited as a basis for test generation techniques. In the chapter concerning ***SDL*** more attention will be paid the the ***ACT***. Than also an informal algorithm for the generation of an ***ACT*** will be given.

2.2.4 Representations

The common representation of finite state machines and labeled transition systems is a directed graph. Each edge in the graph $G(V, E)$ represents a transition, all states of the system are represented by the vertices of the graph. Edges exist for a pair of vertices if a state transition is possible in the model between the two corresponding states. The label that is connected to the edge, represents the relevant action (or input/output pair, in the case of a ***FSM***).

$$G(V, E)$$

- **V** = set of vertices $\{v_1, v_2, \ldots, v_n\}$
- **E** = set of edges $\{(v_j, v_k, l) : v_j, v_k \in V\}$
- **l** = label connected to an edge

The reachability demand in the list of properties of a finite state machine (page 16) implies that the graph has to be strongly connected.

2.2.5 Completion of the specification

Often, the specification for a system defines the behaviour in a normal situation. A normal situation means that the inputs that are offered to the system "make sense". In the state space, such a situation is visible by the fact that for some possible inputs, no state transition is defined (i.e. the δ-function in the ***LTS*** and the λ and δ-functions, that define the (e)***FSM***, are partial functions). However, in section 2.2.3, the transfer function and the output function are defined as complete functions. This means that all combinations $(S \times I)$ (e.g. all inputs in every state) should be defined in the model. If necessary, incompletely specified protocols can be completed by accepting a *completeness assumption*. In the case of a labeled transition system, the model is not necessarily complete. However, it can be completed in a similar way as the finite state model. A completeness assumption can have the form of definition 2.

Definition 2 (completeness assumption) *The system ignores each unexpected input by consuming the input and remaining in its current state without producing an output.*

Such an assumption will add extra transitions to the model that are called *non-core transitions* as opposite to *core transitions*, that are part of the original system. Another way to deal with incompletely specified ***FSM***'s or ***LTS***'s, is to make a distinction between different possible inputs [BEVT89]. There are tree possible input messages in a test system:

- **Valid:** A message that is received by the implementation is *valid* when it was expected by the implementation.
- **Inopportune:** An *inopportune* message is a message that is not expected by the implementation when it is received.

- **Invalid:** An *invalid* message is a message that should never occur. Messages that are outside their bounds or that were incorrectly coded are *invalid*.

These definitions of possible test messages are related to the definition of *core/non-core* edges as follows: *Valid* messages trigger *core* transactions. A completeness assumption like the one in 2, add extra transitions for non-expected inputs. Therefore, *non-core* transitions are triggered by *invalid* and *inopportune* inputs.

2.2.6 Conformance relations

Formally, an implementation conforms to its specification if the *behaviours* of both systems are equivalent. In the literature there are many different definitions of equivalence. An overview can be found in [Kri89, NH84, Koo91]. Defining whether the behaviour of an implementation and a specification is equivalent or not, is a delicate matter. One reason is that they usually exist in different physical domains. An other reason is that the tests are used to decide whether the *functions* of the protocol are implemented without errors. This implies that certain 'different' behaviours, that implement the same function correctly, should both pass the conformance test[7]. According to Holzmann, two systems are said to be equivalent if they *can* generate the same sequence of output symbols when offered the same sequence of input symbols [Hol92]. This means that non-deterministic systems will have equivalent choices in equivalent states. In deterministic systems, each input sequence should lead to the same output sequence for both equivalent systems. Notice that all different forms of equivalence in the list are *equal* for deterministic models. From [P5493], the following definitions are obtained:

- **failure**: deviation from an agreed description of the expected service.
- **error**: that part of the system which is liable to lead to a failure.
- **fault**: the phenomenon that caused an error.

For the definition of some important equivalence relations, a **trace set** is used: A **trace** is defined as a sequence of actions which the system is able to perform, from the initial state $\mathcal{S}_0$. The trace set of a system $\mathcal{S}$ is denoted $Tr(\mathcal{S})$ and contains all possible traces for the system.

- **Trace Equivalence:** two systems $\mathcal{S}_1$ and $\mathcal{S}_2$ are *trace equivalent* ($\mathcal{S}_1 \approx_{tr} \mathcal{S}_2$) iff $Tr(\mathcal{S}_1) = Tr(\mathcal{S}_2)$.
- **Observational Equivalence:** ($\mathcal{S}_1 \approx \mathcal{S}_2$) iff
for every action α, $\mathcal{S}_1$ is able to perform, there exists a successor of $\mathcal{S}_2$ via α

[7]It will be shown later that the notion of equivalence could differ for other purposes, like verification.

such that the resulting behaviours of $\mathcal{S}_1$ and $\mathcal{S}_2$ are *observational equivalent* , and similarly with $\mathcal{S}_1$ and $\mathcal{S}_2$ interchanged.

- **Testing Equivalence:** two systems are *testing equivalent* ($\mathcal{S}_1 \approx_{te} \mathcal{S}_2$) iff $\mathcal{S}_1$ and $\mathcal{S}_2$ *may* and *must* satisfy the same set of observers [NH84].
  - a system $\mathcal{S}$ *may* satisfy an observer $\mathcal{O}$ if a successful test exists
  - a system $\mathcal{S}$ *must* satisfy $\mathcal{O}$ if all tests are successful
- **Failures Equivalence:** failure equivalence resembles testing equivalence [FB91, LBDW91]. If failure equivalence is required, the set of traces of the specification shall be included in the set of traces of the specification, and the implementation shall not produce unspecified deadlocks [P5493].

The relation between these notions of equivalence is:

$$\mathcal{P}_A \approx \mathcal{P}_B \Rightarrow \mathcal{P}_A \approx_{te} \mathcal{P}_B \Rightarrow \mathcal{P}_A \approx_{tr} \mathcal{P}_B$$

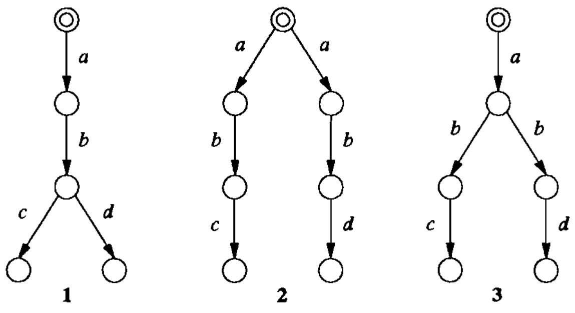


Figure 2.8: Equivalence between systems

Let the graphs in figure 2.8 represent a part of the behaviour of three systems. The systems start in the up-most (double circled) state where they are able to execute an action a. All systems are trace equivalent since their trace sets are equal:

$$Tr(\mathcal{S}_1) = Tr(\mathcal{S}_2) = Tr(\mathcal{S}_3) = \{<>, < a >, < ab >, < abc >, < abd >\}$$

Non of the systems is *observational equivalent*. Each system contains a state in which the system does not have the same choice as the other systems. For instance, after having executed action a, $\mathcal{S}_2$ no longer has a non-deterministic choice between the sequences $< bc >$ and $< bd >$. Since the rest of the behaviour of $\mathcal{S}_3$ is not yet determined after action a, $\mathcal{S}_2 \not\approx \mathcal{S}_3$.

As systems $\mathcal{S}_2$ and $\mathcal{S}_3$ both may refuse an action c or d after the second transition, the observer that *may* be satisfied by $\mathcal{S}_1$ differs from the one of the two other systems.

Therefore, $\mathcal{S}_1 \not\approx_{te} \mathcal{S}_2$ and $\mathcal{S}_1 \not\approx_{te} \mathcal{S}_3$. From the outside no observer can distinguish between systems $\mathcal{S}_2$ and $\mathcal{S}_3$ so $\mathcal{S}_2 \approx_{te} \mathcal{S}_3$.

In practical cases, the observational equivalence may be to strong to prove. In the case that tests are derived from LOTOS specifications, failure equivalence is often used. Trace equivalence is often used for ***SDL***-based test generation techniques. In [SD88, Hol91a, Ura91] difference is made between *strong* and *weak* conformance. Notice that for models in which a completeness assumption like in definition 2 is already implemented, there is no difference between weak and strong conformance.

- **Strong Conformance:** If the implementation mimics the behaviour of the specification for every transition (*core* and *non-core*) the two have strong conformance.
- **Weak Conformance:** An implementation has weak conformance to its specification if the implementation generates the same outputs for each sequence of *core* transitions. The behaviour may differ for *non-core* transitions.

2.2.7 Testing strategy

Most existing test techniques have the same structure. Assuming there is a suite of tests available, the tester will perform the following actions:

- go to a certain state;
- apply an input to the system;
- receive outputs from the system;
- compare the received outputs with the expected outputs;
- check if the expected state is reached (optional).

If the received outputs do not match the outputs that can be expected on the basis of the specification, the verdict of the tester will be a `fail`. However, when the outputs match the expected outputs the situation is more complex. First, assume that there is no nondeterministic behaviour in the specification. In this case, one would be tempted to verdict a `pass` for that part of the system. It is, however, always possible that the system reacts with different behaviour the next time a similar trace of actions is executed. It is not possible to test each action an infinite number of times. In order to able to conduct a meaningful conformance tests, the test person must accept the assumption that no such behaviour is implemented in the system.

In the case that nondeterministic behaviour can be expected (i.e. the case that the model does not hold enough information to determine what choices will be made), the tester should at least examine every possible choice. The strategy, therefore, will be to execute every possible behaviour at least once. This can be done by executing a certain sequence of actions repeatedly. In most cases the nondeterministic behaviour is actually deterministic behaviour that depends on parameters that were not in the model. It is the task of the designers of the tests, to include such parameter ranges and data values that every possible behaviour is executed. Proving that the implementation under test contains errors is generally far more easier that proving it is correct.

Another problem that is presented to the test designer is that it is very hard to determine whether spontaneous outputs will follow or not. On the one hand, if not all outputs have been generated by the system, the tester should wait until all outputs have been generated. On the other hand, the tester must ensure that only the expected outputs are generated and nothing more. In both cases, the tester must determine if more outputs will follow or not. In practice, this problem is solved by defining (heuristic) timer values. Without good knowledge of the implementation under test, this may be a difficult problem.

2.3 Composition of finite systems

In many cases, the system that is to be tested consists of more than one process. The existence of multiple processes in a system means that there are different entities that operate parallel next to each other. These processes may exchange messages and data. Situations in which multiple processes reside in a system that is to be tested are for instance [DKK91]: implementations that consist of more than one protocol or protocol layer. If the manufacturer has integrated two systems into a single implementation, only the resulting composite system can be tested. It may also be necessary to test a system in a part of its normal environment. In this case, the system under test is the composition of the system and its environment. Multiple processes may also occur in a specification of a system. The formal language ***SDL*** is an example of a language that supports the definition of many different processes in the same system.

An external observer is only able to see the behaviour that occurs at the boundary of the total system. The system that is seen by this external observer is called the **composite system** and the possible behaviour of this system, the **composite behaviour**. The composite behaviour is determined by two aspects:

- the specification of the separate processes;
- the communication between the processes.

The communication between the processes in an important factor. Two kinds of communication can be distinguished: **synchronous communication** and **asynchronous communication**. In the first form of communication, messages are exchanged by negotiation. This means that the sending process waits until the receiving process is ready to receive a message, then in an infinitesimal amount of time, the message is exchanged after which both process proceed.

The second form of communication, asynchronous communication, assumes that each receiving process has a queue associated to it. The sending process may put the message in this queue at any time[8]. The receiving process takes messages from the queue and processes them.

For the automatic generation of tests from formal specifications, a model of the composite state space (i.e. the state space of the composite system) is necessary. This model should consist of all states and the actions that may cause a transition to a new composite state. Many of the state transitions may be invisible at the outside of the system, since they represent internal communications between two processes in the system. The state space of a system specified as multiple processes is composed of the Cartesian product of the state spaces of all processes. In the case of asynchronous communication, each state in the composite state space also holds the state of the queues in the system:

$$S = S_1 \times S_2 \times \cdots \times S_n \times Q_1 \times \cdots \times Q_m$$

The system may transfer from one state to a new state by executing an enabled action. In the resulting model, three kinds of (composite) states can be distinguished:

- **non-reachable states**: States that can not be reached by any combination of stimuli at the boundary of the system.
- **reachable, non-stable states**: States that can be reached by the system from its initial state. If a state is non-stable, the system may transfer to an other state without any stimulus from outside the system.
- **reachable, stable states**: A stable state is a reachable state of the system. If the system has reached that state, it will only leave that state if it receives an input-signal from the environment. In the rest of this report, the expression "stable states" will be used to denote reachable stable states.

For all reachable states in the system it will be assumed that: *any reachable state can be reached from every other reachable state in the system.* This means that there has to be some kind of periodicity in the system. This assumption can easily be verified in practical systems, which usually have a "reset"-like function.

For an external observer, it may not be possible to determine whether an internal communication is finished or not. Therefore it may also be impossible to execute actions in non-stable states. Therefore, when testing a system that consists of multiple processes,

[8]Sometimes, an upper bound for the queue length is specified. In this case the sending process may have to wait or the message may be lost, depending on the way the queues are defined

usually only a part of the reachable states is tested. The tests are executed by applying an external input to the system. After this the tester waits long enough until it is reasonably to assume that all enabled internal actions have been executed. In later chapters, much more will be said about the testing of composed systems.

2.4 Testability

Testing protocol implementations is a difficult task. During the past years, much work has been carried out to improve methods for automatic generation of conformance tests. Also, the standardization organizations like ISO, ETSI and CCITT have been advocating conformance testing methodologies and the use of formal languages. Still, the practical use for conformance testing is still limited to medium-sized protocols [DSU90, Hog91b]. There is a number of reasons that make the generation of good conformance tests for large and complex protocols [LL92] a difficult task or even impossible. An important reason for this was already mentioned: the limited observability and controllability. The limited amount of time that is available to execute the generated tests poses an other limitation. A number of general problems that limit the possibilities of conformance testing will be summarized in this section.

2.4.1 Nondeterminism

The testability of systems (i.e. the possibilities that exist to test the systems) is worse when the system inhibits nondeterministic behaviour. Some approaches when testing nondeterministic systems were already presented in the section about testing strategy. Nondeterminism that results from abstraction of parameters or variables can be removed by choosing relevant data and parameter values. However, nondeterminism can also result from behaviour that can not be influenced by the tester such as critical races or erroneous channels. In this case, repeatedly testing is the only solution.

2.4.2 State space explosions

The complexity of the protocol pushes all existing test techniques to the limits. An average protocol can already inhibit very large numbers of states. Each state and all possible state transitions must be considered when creating tests. Especially, when translating descriptions of protocols that are written higher-level description languages to lower-level models like ***LTS***'s or ***FSM***'s, the number of states rapidly grows very large. This phenomenon is called the **state space explosion**. There are two main reasons for the explosion of the number of states.

The first reason for the state space explosion is the existence of variables in the description of the model. In a lower-level model, each value of such a variable represents a new state. Queues have a similar effect on the state space, since each contents of a queue represents different behaviour, and thus a different state. When queues are not bounded or when a variable can hold an infinite range of values, the number of states may even become infinite.

A second reason for the state space explosion is the fact that concurrent processes in the description are composed into one single description. This means that every possible interleaving of concurrent actions will be present as a unique path in the resulting model (this part of the state space explosion is also called the **combinatorial explosion**).

The enormous amount of states that are held in the resulting model limit the possibilities of testing. First, the generation of tests will become very difficult. The system will become to complex for manual generation of tests and the model will become to large for the resources that are available nowadays. Secondly, if one succeeds in generating tests, the number of tests will be far to large to use for a practical conformance test.

2.4.3 Fault coverage

The limitations for conformance testing raise the question of what the reliability of a conformance test will be. The reliability is expressed as the **fault coverage** of a test, which is the chance that possible errors in the specification are detected by a test. The fault coverage can never be a full 100% for practical systems. However, for some systems, conformance tests can be generated that can obtain a very high degree of certainty.

Generally the limited amount of time that is available to perform the tests is the major bottle neck in achieving very high fault coverages. Also the choices that are made by the test person for the relevant data values and parameters is of very large influence for the fault coverage of the conformance test.

2.4.4 Design for testability

It is generally recognized that a part of the solution toward the problems; that have been presented in this section lies in the specification. If a protocol is specified with conformance testing in mind, large improvements can be achieved. ETSI is standardizing rules for specifying protocols such that testing and validation is facilitated in later stadia [ETS92]. In the SPECS project (RACE 1046), a lot of attention was paid to the design of the specification [BKP92].

Independently of the formal description techniques that is used, a number of improvements to the specification can be made. This way, the testability of a specification is increased substantially [EK92]:

- the definition of an upper-bound for the queue lengths;

- the definition of upper and lower-bounds for parameters and variables;
- the definition of a maximal system response time;
- the definition of extra Points of Control and Observation;
- adding extra functionally to the system that facilitates testing.

The last two items lay some of the responsibility for the testability with the manufacturers. In other fields, this is not new. For instance, large integrated circuited nowadays often are equipped with boundary scan architectures that allows the IC's to be directly accessed, even after they are mounted. Extra ***PCO***'s could give the tester the possibility to improve the controllability and observability by being able to access internal variables directly. Extra functionality could also serve for extensive status messages, complete reset functions and direct state transfers.

3 Automatic test generation

Manual test generation is based on a good insight in the relevant protocols and the possible parts in which errors may reside. However, with the growing complexity of protocols, this insight is no longer sufficient. Manually generated tests often do not test all possible behaviour, which has a negative effect on the fault coverage. Besides the technical problems of designing good conformance tests, there is also the problem that two conformance tests, generated manually by two different designers, will be dissimilar. This is a major disadvantage when trying to standardize the conformance tests and the way they are used. In practice, this will mean that several conformance tests are (unnecessarily) conducted on the same product.

Automatic test generation seems to be able to obviate some of the disadvantages of manual test generation. Providing complete and correct formal models of the protocol are available, automatic test generation can be used to generate tests that:

- achieve a high level of fault coverage;
- are errorless;
- are the same, wherever they are made;
- can be made relatively fast.

This chapter describes some existing test generation techniques. Starting point of most of these techniques is a finite state-model, that was described in the previous section. However, also a short description of other possible test generation techniques is added. The chapter is completed with a short description of the automatic test generation tool the PTT Conformance Kit.

3.1 Introduction

Most automatic test generation techniques are based on finite state descriptions of the protocol. The reason for this partially lies in the fact that this model formalizes many properties of black-box conformance testing as described in the previous chapter. Typical properties that distinguish the ***FSM*** model from other representations are the finiteness of the model, the absence of nondeterminism, and the state transitions that are based on input/output pairs.

input/output pairs

The basic strategy of black-box conformance testing is to apply an input to the implementation, then wait until all outputs from the ***IUT*** have been generated, and finally check whether these outputs conform to the specification. This typical black-box testing strategy is illustrated in figure 3.1. The states of the protocol that are considered are states in which an input is expected from the environment, just as in the ***FSM*** model.

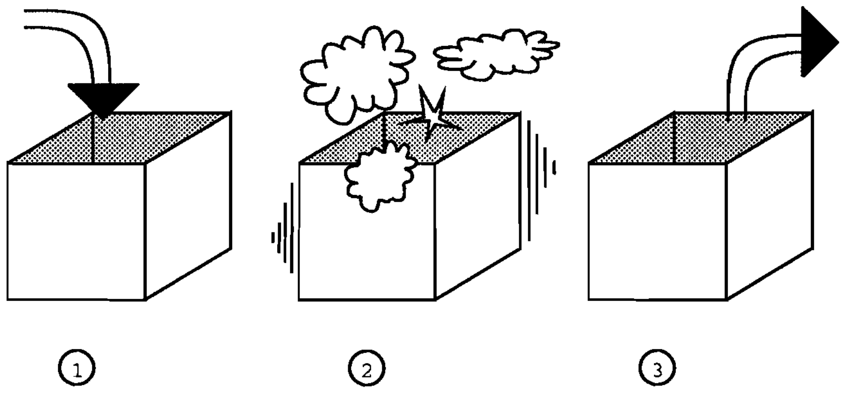


Figure 3.1: Basic concept of black-box testing

finiteness

A system that is modeled as a finite state machine should consist of a finite number of reachable states and a finite number of state transitions. In practical systems, ranges of variables and the lengths of queues always are bounded, so this demand is usually met. While it may be difficult to conform to this demand when translating a formal description of a protocol into an ***FSM***, it is no limitation from a testing point of view.

nondeterminism

Often, formal descriptions of a protocol allow for nondeterministic behaviour. In the physical systems that are considered here, there is no such thing as nondeterminism. There may however be behaviour that can not be controlled by the tester. The easiest way to deal with such behaviour is to treat it as if it were nondeterministic. If enough information is available about the protocol implementation most of the nondeterministic behaviour can be removed from the model. The removal of nondeterminism from the model is, however, not always possible so special care should be taken with this property of the finite state model.

3.2 FSM-based test generation techniques

All conformance tests that are derived from a ***FSM***-model will have the same global structure: First apply an input to the system, then check whether the outputs that are generated conform to the outputs that would be expected from the specification. There are, however, some major differences in the way various methods implement these steps. An overview of these methods is presented in [Ura91]:

random walk

A simple approach to automate test generation is to apply *random* sequences of inputs to the ***IUT***, until every transition was tested or until it is reasonable to assume that a sufficient part of the total behaviour is tested. The system will be forced to "walk" through its state space, while the tester is scanning for errors. It can be observed [LCL87] that an error in the implementation is often repeated several times in the state space. The random walk method, therefore, is a good alternative to the more complex test generation methods based on non-random walks through the state space.

transition tours

A different approach is to guide the system in a walk through the total state space. This way, it can be verified that every possible transition is executed at least once. The first method that is described here is the **transition tour** method. Here, the test generator *tours* through the complete state space. When the start state and the end state of the tour are equal, such a sequence of inputs is called a transition tour.

Definition 3 (Transition Tour) *A* transition tour *of M is an* input *sequence which takes M from its start state S_0, executes every transition of M at least once, and then returns M to state S_0 [Ura91].*

The efficiency of the method can be improved by minimizing the length of the transition tour. In graph theory the problem of finding such a minimal tour is known as the Chinese Postman Problem [ADLU88]. For most systems a minimal transition tour (i.e. the shortest test sequence that covers all state transitions of the specification) can be computed.

An important disadvantage of the transition tour method is that it lacks the possibility to check whether the tour actually reaches the intended states. An other problem occurs when an error is found. Since the new state of the system after an error is not known, the tour can not be completed after an error is found. So, once an error is detected in the implementation, the rest of the generated test is useless. If the tests are part of a implementation/debug cycle, only one (the first) error can be fixed at the time.

There is a number of methods that are able to continue a meaningful test after detecting an error. Most of them yield a set of independent test cases. Such a set is known as a"*partitioned tour*". The structures of these methods are very similar. For each possible transition:

1. Bring the system to a known starting state with a "*synchronizing sequence*";
2. Bring the protocol implementation to a certain state (a sequence to transfer the system from any state to a state s_i is a *transferring sequence* for s_i, TS(s_i));
3. Check one state transition by applying the subsidiary input and check whether the appropriate output is generated (this is called the *checking sequence*);
4. Check whether the system has reached the intended new state.

The only difference between these methods is the way in which the last step is performed. One of the possibilities to check whether a system has reached a certain states is the *status message*. A status message could be implemented as a special transition that outputs status information and stays in the same state. In such cases, the functionality that is necessary must be available in the specification. Normally, however, this is not the case, so different methods are necessary to determine whether the correct state was reached.

distinguishing sequences

A **distinguishing sequence** (***DS***) is an input sequence for a protocol such that the outputs generated by the implementation will identify the state of the protocol implementation.

Definition 4 (Distinguishing Sequence) *An input sequence is a* distinguishing sequence *of* FSM *M if the output produced by M in response to the sequence is unique for each state of M [Ura91].*

The advantage of this method is that information is available about the end state of the erroneous transition, presumed there are not to many errors. Unfortunately, for most practical protocols, there are no distinguishing sequences, and if they exist, they might be very long.

characterizing sequences

A set of **characterizing sequences** [Cho78] serves the same purposes as a distinguishing sequence. They both answer the question: "*What is the current state of the implementation?*". A characterizing sequence (***CS***) is defined on a subset of states. A complete finite state machine that contains no equivalent states always has a set of characterizing sequences that distinguishes every state of the ***FSM*** [Ura91]. For such an ***FSM*** the definition is:

Definition 5 (Characterizing Sequences) *A Characterizing sequence is an input sequence which output distinguishes two or more subsets of states.*

A *set* of sequences which can distinguish the behaviour of every pair of states in an ***FSM*** is called a *characterizing set* (W-set). Such a set is applied in the same way as a distinguishing sequence. Test sequences that are generated in this way are usually longer compared to other methods.

unique input/output sequences

In [SD88] a new test generation method was suggested that uses *Unique Input-Output sequences* (***UIO***s). ***UIO***-methods only determine *if* the expected state is reached. Instead of generating a single sequence that identifies the current state, an ***UIO*** sequence is generated for each state of the state machine.

Definition 6 (Unique Input/Output sequences) *A* UIO *sequence for state s_i is a specific input/output sequence with the originating state s_i such that there is no $s_j \neq s_i$ for which the sequence is a specified input/output sequence (i.e. from which this sequence could originate, according to the specification) [SD88].*

To limit the length of the test that is generated the algorithm generates *minimal* ***UIO***-sequences i.e. a ***UIO***-sequence that does not contain any other ***UIO***-sequence. An algorithm could also check if the generated test-sequence (***TS***-transition-***UIO***) is contained in, or contains, an other test-sequence. Further optimization of this method can be achieved by using the ***UIO***v, ***SUIO***, or the ***MUIO*** method:

- the ***UIO***v-method ensures that all ***UIO***-sequences are also unique for an ***FSM*** that contains errors.
- In the ***SUIO***-method [ADLU88], a *rural Chinese postman tour* is generated to find the shortest connections between test segments.
- For a model, more than one ***SUIO***-sequence may exist. The ***MUIO*** method searches for the shortest ***SUIO***-sequence.

3.3 Tree-based test generation techniques

When modeling the behaviour of a protocol, it is not always possible to obtain the state space from the specification. For instance, there may not be enough memory in the computer to hold every possible state. A different approach to the modeling of the system's behaviour is to construct a behavioral tree. The root of the tree will represent the starting state of the system. Each path through the tree represents a legal sequence of actions. A behavioral tree can be obtained from a state graph by unfolding it and leaving out the state names. A tree that models a typical communication system is *infinite* (i.e. it never "stops"). Some algorithms, however, do check whether a (composite) state has already appeared in the tree, in which case the tree branch does not have to be extended. However, this eliminates the advantage of not having to store the complete state space.

Test generation from a behavioral tree can be done by testing the paths in the tree. The generation of tests can be done "on the fly" so there is no necessity to hold the complete state space model in memory. The major problem of this approach is that there is no general overview of the global behaviour (e.g. the (composite) state space as in the ***LTS*** and ***FSM***-models). Therefore it is very difficult to limit the size of the tree (and thus of the generated tests).

An important application of the behavioral tree is the modelling of the observable behaviour of ***SDL***-specified systems. Hogrefe [Hog88, Hog90] presented some work on the automatic generation of tests from ***SDL***-specifications using an **asynchronous communication tree** (***ACT***). The ***ACT*** models the observable behaviour of the system. The theory of the ***ACT*** was first presented by Agha [Agh86] and is also used in the project "ARTEFACT" [DGK91] at PTT Research and the RACE project on computer aided test generation CATG [PRO92].

3.4 Test generation based on canonical testers

A very formal approach to conformance testing is the use of canonical testers [Bri88]. A canonical tester is an environment process that exchanges messages with the ***IUT***. The theory was developed for the description language LOTOS. In theory, a canonical tester is able to perform an exhaustive test (i.e. all possible behaviour is tested). In practice, however this means that tests generally will not terminate within finite time. The *practical* use of these methods is relatively small. The theory of canonical testers is beyond the scope of this report.

3.5 PTT Conformance Kit

The **PTT Conformance Kit** is a set of tools that is meant to help with the generation of conformance tests [DGK91]. The basic idea was that as large a part of the work that could be automated would be covered by the toolkit. It is recognized, however, that the interaction with the people who are conducting the tests, is essential.

It is assumed that the specification is modelled as an extended finite state machine (***EFSM***). This model extends the ***FSM*** model with variables and predicates. There are also tools available to compute the behaviour of multiple, interacting ***FSM***'s. The general set-up of a test generation procedure is:

1. obtain an ***EFSM*** specification of the protocol. The **Conformance Kit** will check the semantics of the specification;
2. simulate the specification to validate its behaviour
3. design a test suite structure. In this step parts of the behaviour can be excluded and the appropriate test types are selected;
4. automatically generate the desired tests (all generated tests are written in the TTCN-MP[1] format);
5. Create the constraints part to form a complete test suite;
6. If necessary, convert the generated tests to graphical TTCN, this representation is especially suitable for human interpretation.

The methods that are available for test generation are: **Transition Tours**, **Unique I/O sequences** and **Random Sequences**. The former two are already described in other sections. The latter, **Random Sequences**, generates tests that check randomly selected parts of the behaviour. Typically, the complete tests that are generated with the Conformance Kit have the form of a **Partitioned Tour** (see page 30). Simple tools are also added that facilitate the testing of data aspects of the specification. Currently, work is in progress to add important new modules to the Conformance Kit that enable the generation of data tests.

[1]Tabular and Tree combined Notation; Machine Phrase format. This notation is standardized by the ISO.

In the specification, the test designer may also mark some behaviour with a label. These labels can be used when generating tests with the Conformance Kit, to exclude some behaviour from the tests. This feature can be used to select only a subsection of the possible behaviour of the systems for test generation. This is an important aid in dealing with the complexity of the protocols.

The Conformance Kit was used in different projects at the PTT Research Laboratories. One of these projects was the development of test suites for ISDN terminals [DHN90]. In this project, the toolkit was used to generate a test suite suited for the local test method. With the aid of other tools these tests were adapted for a remote testing environment. This was done by computing the composite behaviour of the system and the test context.

One of the problems when using the algorithms that were implemented in the Conformance Kit is the state space explosion, which limits the applicability of the methods to medium-sized protocols. Various methods have been proposed to improve the ability of the algorithms to be applied to very large protocol specifications. In short these methods can be grouped in three categories [Hol87]:

- **reduction**: Exploiting equivalence to reduce the number of states or transitions that must me explored;
- **truncation**: Making a selection in what is analyzed in a protocol. These methods are generally based on heuristics;
- **efficacy**: By improving the efficacy of the algorithms large improvements can be made.

The efficacy of an implementation of one of the presented algorithms can make a lot of difference. In [Hol89], large improvements are reported with methods that do the storage of state information in a more efficient way. This subject is not within the scope of this report. Truncation and reduction techniques both try to decrease the size of the protocol specification. The use of labels in the Conformance Kit is an example of the use of truncation in dealing with complexity. Methods for both techniques will be presented in the following chapters.

4 SDL specifications

Recently, the formal language ***SDL*** (Specification and Description Language) has received a lot of attention. In contrast with other, often more academic, formal languages the use of ***SDL*** in specifying complex systems is growing. One of the reasons for this (relative) popularity of the language is that is was designed with the designer of a protocol in mind. ***SDL*** was standardized by the CCITT and was also accepted by the ISO. In the standard of ***SDL*** [IC92] the objectives of the language are stated as:

> The purpose of recommending ***SDL*** is to provide a language for unambiguous specification and description of the behaviour of telecommunications systems. The specifications and descriptions using ***SDL*** are intended to be formal in the sense that it is possible to analyze and interpret them unambiguously. The general objectives when defining ***SDL*** have been to provide a language that:
>
> - is easy to learn, use and interpret;
> - provides unambiguous specification for ordering and tendering;
> - may be extended to cover new developments;
> - is able to support several methodologies of system specification and design, without assuming any particular methodology.

The purpose of this chapter is to introduce some general concepts concerning ***SDL***, such that test generation techniques that are based on ***SDL***-specifications can be presented. For this purpose, also some representations of the composite state space of an ***SDL***-specified system are presented. Starting point is **basic *SDL***, which is a subset of all possible structures in the language. At the end of 1992, the latest revision of the Z.100 international standard was accepted by the CCITT [IC92]. Some important changes have been carried through. However since all literature up till this moment has been based on the 1988 version of the language, SDL'88 will be the basis for this report. Wherever features of the 1992 version are used, this will be stated explicitly.

4.1 Structure of SDL specifications

In ***SDL***, both a phrase-syntax and a graphical syntax are available for the designer. Both syntaxes can be used to describe the same behaviour. However, since the graphical form is more illustrative, the latter will be used in this report. In an ***SDL***-specification, three hierarchical levels can be distinguished. The first is the total *system*. This system is build from one or more *blocks*. Communication between the different blocks in the systems is done via *channels*. Each block can be divided into multiple sub-blocks. The partitioning of the specification into different refinement levels is an important aid to tackle the complexity of the system.

Inside a block, at least one process is present. A process is defined as an extended finite state machine that runs parallel with respect to the other processes. Processes can exist at startup or can be created and deleted during run-time. A state in an ***SDL***-process represents a state of the process in which it waits for an input to arrive at its input queue. Building blocks of a process are: inputs, outputs, decisions, process creation/deletion and tasks. The definition can be extended by parts of the total set of ***SDL***-constructs. Only the SAVE-construct will be described here.

The various *processes* that exist in the blocks at the lowest level communicate with each other and the borders of the block via *signal routes*. Figure 4.1 gives an example of an ***SDL***-system containing two blocks and three processes. The thick lines between the blocks denote communication channels, while the thin lines inside the blocks represent signal routes.

The subdivision of the specification into hierarchical levels is not necessary to explain the basic concepts of test generation for systems that are specified in ***SDL***. Therefore it will be assumed that the specified system only consists of different processes that communicate with each other and with the environment via channels i.e. no blocks are being considered.

Signals in the simplified system are transferred via channels asynchronously. This means that the sending process does not have to wait for the receiving process to accept the message. Each process is assigned an input queue for all arriving communication channels. In these queues, the signals are stored in ***FIFO*** order (see figure 4.2). During the transport over the channel, the signals suffer a non-deterministic but finite delay[1]. As a result, two signals that where sent at the same time to the same point but via different channels, may arrive at their destination in completely arbitrary order. For the communication between the protocol implementation and the tester (i.e. the environment) the same mechanism as inside the system is assumed.

[1]In the most recent version of ***SDL***; SDL-92, the delay may be specified as zero for a channel.

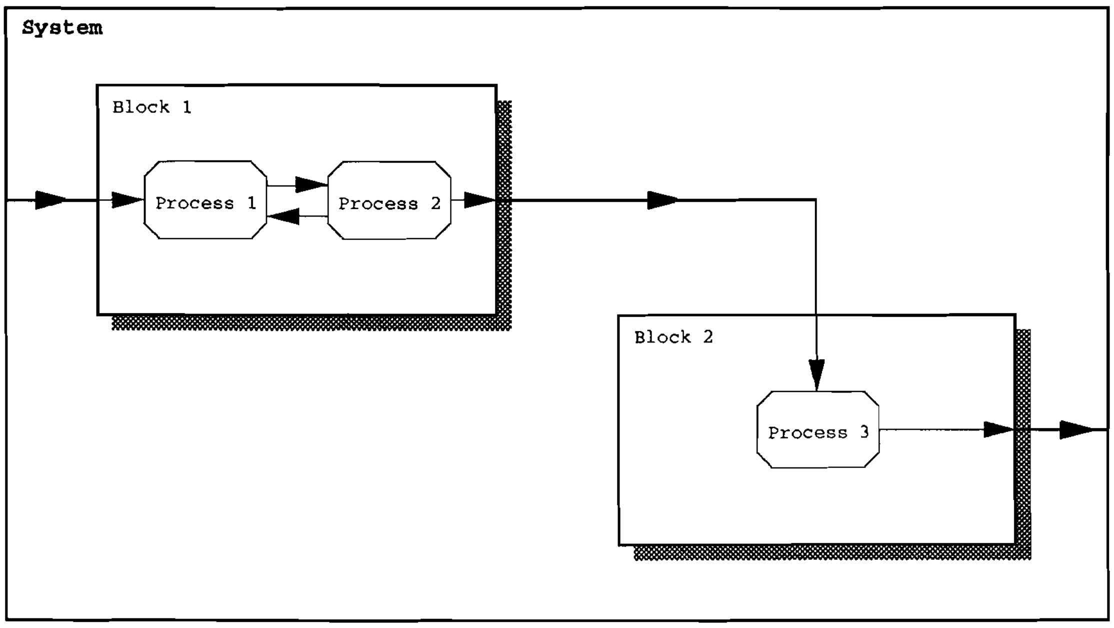


Figure 4.1: Basic structure of an SDL system.

4.2 Observable behaviour

In a testing-environment it is impossible to obtain information about the system by any other means than its behaviour at the outside. This is also the only place at which the tester can influence the state or behaviour of the system. Usually, the interface between the system and its environment is well defined. Such a place is called a Point of Control and Observation (***PCO***). At a ***PCO***, signals leave or enter the system. Depending on the kind of protocol that is tested, the tester communicates with the system directly or via a communication medium (that could include, for instance, other parts of the protocol stack). In the rest of this report it will be assumed that the communication between the ***IUT*** and the tester is error-less and transparent so the tester will act as the environment of the protocol.

Clearly the reception and sending of signals via the ***PCO***'s by the tester are observable by the tester itself. The sending and reception of signals by the environment actually consist of two actions: a **send**-action, which puts the message on the channel and a **receive**-action, which reads the message from the input queue[2]. Yet, the systems behaviour does not change until the reception of the pending signal. Therefore, only the

[2]Actually the arrival of the signal and its insertion in the queue can also be distinguished. However, there is no way of observing these events separately from the outside the system

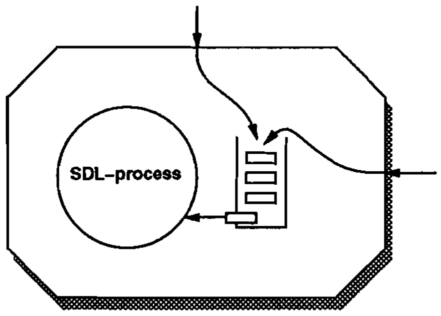


Figure 4.2: Input queue of an SDL-process.

reception of the signal will be denoted as "observable" (i.e. the action of the receiving process that reads the signal from the processes input queue) as the action of the system that can be observed by the environment.

As in the case of communication between the system and the environment, an internal communication between two processes in the system is composed of a **receive** and a **send**-action. But in this case, neither of those can be observed at the outside of the system directly. The most straight-forward approach would be to mark those actions as 'not observable'. It will be shown that establishing a satisfactory conformance relation between the protocol specification and its implementation is very difficult on this basis. Although the internal communication itself is not visible at the boundaries of the system, it should be represented in the model since the results of it are very likely to become visible in the future. Therefore the model for the external behaviour of the system will be extended with a τ-action. This τ-action represents the reception of an internal communication by the receiving process. Summarizing, the observable behaviour of a system specified in ***SDL***is defined by:

- reception of a signal by the environment;
- reception of a signal that was sent by the environment;
- reception of an internal communication (the reception of a signal by a process in the system, that was sent by an other process in the system);
- in implicit transition, caused by the reception, either from an other process or the environment, of an illegal signal.

4.3 Nondeterminism

Almost all existing test generation techniques are based on the assumption that the protocol implementation is deterministic and therefore behaves in a deterministic way.

There are however various ways in which nondeterminism can be introduced into a system model.

through the use of SDL

The 1988 version of the language itself had no constructs to introduce nondeterminism directly. However nondeterminism may arise from the communication mechanism. Signals may be sent while other messages are pending and there is no way to determine which signal may arrive at its destination first, leaving the future behaviour nondeterministic.

Under pressure of protocol designers, the language has been extended with special constructs to model nondeterminism in the 1992 version of ***SDL***. The following constructs have been added for this purpose:

- **spontaneous transition** A spontaneous transition allows the activation of a transition without any stimuli being presented to the process;
- **nondeterministic decision** A decision block may decide on the basis of a "question expression". In ***SDL***'92, the question expression may also be **any**, denoting that any of the possible choices may be selected by the system nondeterministically;
- **anyvalue expression** A random value for a variable may be selected by the expression: `var` := **any**`(<sort>)`.

through abstraction

State of the art communication protocols usually inhibit such an amount of complexity that abstraction from technical details is necessary to be able to build the internal representation that is necessary to generate tests. Possible abstractions are "if reduction" (see figure 4.3) and neglecting parameter and data values. In ***SDL***, the question expression may also be put in informal text. In this case, the designer is forced to perform an if-reduction step.

There are two possible approaches that are usually combined in the modeling of a ***SDL***-specified system.

- Every choice in a process is based on known parameters and data that is read from inputs. For every legal input, some behaviour is defined which may be dependent of the values in the data-fields of the input. Each value or range of values in the data-field of an input may be defined as a different input. For instance, a single input of an integer with a legal range of [1..10] would result in 10 different inputs.

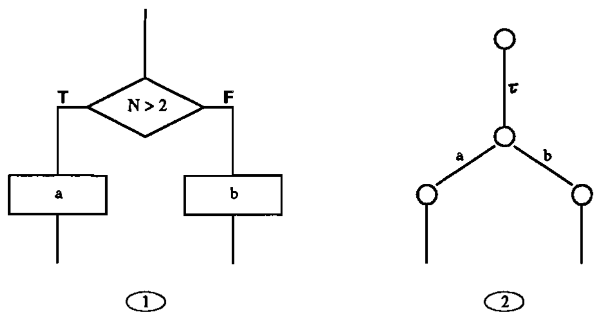


Figure 4.3: Model of the state space of a part of the *SDL*-specification in (1) after "if reduction".

- A second way is to make an abstraction and model choices on the basis of variables nondeterministic choices. This way of modelling is called "*if-reduction*". Introducing nondeterminism like this makes the model less accurate, but is sometimes necessary to limit its size.

through unspecified options

Protocol specifications often define large families of protocols, which are each characterized by their own set of options. One way dealing with those options is leaving the choice between the different kinds of optional behaviour nondeterministic. However, since conformance tests usually only are made for specific implementations, of which the options are well known, this kind of nondeterminism usually is not necessary.

Dealing with nondeterminism in the context of conformance testing is an important problem. There are two aspects to the problem. The first is a definition of an appropriate equivalence relation according to which it could be determined whether one system conforms to the other even when the specification is nondeterministic. The other aspect is the fault coverage of the resulting test sequences. Since a nondeterministic system can not be put in any arbitrary state directly, test cases may have to be repeated several times. Even when test cases are run many times, there will always remain a chance that some part of the behaviour is not tested.

4.4 Modeling the state space of SDL-specified systems

It was already mentioned that a specification of a protocol in a formal language often is not a good starting point for automatic test generation algorithms. The general approach, which will be followed here, is to create an intermediary model of the protocol that holds information about the composite state space of the system. This composite state space is composed of the state spaces that belong to the separate processes. Each composite state represents some future behaviour of the system. To do so, a composite state is build from a local state for each competing process, completed with the pending messages. Two different approaches can be distinguished. The first approach uses a complete model for the composite state space. After the generation of tests, the non-observable actions are removed. In the second approach, which is represented by the ***ACT***-based methods, first a model is generated that only contains the observable behaviour. From this model, tests can be generated directly.

4.4.1 Finite state machines

Almost all methods for automatic test generation use finite state machines as base model. A state in an ***FSM*** corresponds to a stable state of the system. A stable state is a state in which the system can not perform any action but the consumption of an external input signal. A transition from a state is marked with the input signal, combined with the outputs that are sent to the environment until the system reaches the next stable state. There may be zero outputs to the environment (a silent transition). In the ***FSM***-model, the number of reachable states is finite. For each state in the ***FSM***, the output behaviour to each input is deterministic and known.

The finiteness of ***FSM***s poses a problem when modeling ***SDL***-specifications since a system that is specified in ***SDL*** may be infinite. This is a result of the fact that the number of processes may in theory be infinite and the queue-lengths are not bounded. It is however questionable whether this is a necessary property when specifying "real" protocols. When a protocol is designed carefully, this does not have to be a problem.

The nondeterministic behaviour of a system that is specified in ***SDL*** is however very difficult to model in a finite state machine. If finite state machines are to be used as a model for the state space of a system specified in ***SDL***, adaption of the semantics of ***FSM***s are necessary.

4.4.2 Labeled transition system

In protocol verification techniques, a commonly used model for the composite state space is the **labeled transition system** (***LTS***). In an ***LTS*** a state transition is possible from a composite state for every action that can be executed by each of the processes in that composite system. A composite state in the ***LTS*** exists for each reachable combination of local states and pending signals. The ***LTS***-models inhibits every possible action, observable and non-observable and is not necessarily finite.

The state space of a system specified in ***SDL*** is composed of the Cartesian product of the state spaces of all processes and all possible states of the queues in the system:

$$S = S_1 \times S_2 \times \cdots \times S_n \times Q_1 \times \cdots \times Q_m$$

The system may transfer from one state to a new state by executing an enabled action. The transitions and the state space together form the labeled transition system. In the ***LTS*** three kinds of states can be distinguished:

- **non-reachable states**: States that can can not be reached by any combination of stimuli at the boundary of the system.
- **reachable, non-stable states**: States that can be reached by the system from its initial state. If a state is non-stable, the system may transfer to an other state without any stimulus from outside the system.
- **reachable, stable states:** A stable state is a reachable state of the system. If the system has reached that state, it will only leave that state if it receives an input-signal from the environment. In the rest of this report, the expression "stable states" is used to denote reachable stable states.

The asynchronous nature of the communication in the system poses some difficult problems. Each communication actually consists of two actions: The first action is the output-action of the sending process, which puts the signal on the communication channel. The second action is the consumption of the message from the input queue of the receiving process[3]. If the signal passes through a channel (i.e. a communication between two ***SDL***-blocks or communication with the environment), it will suffer a non-deterministic delay. As a result of this, the sequence in which outputs are received, is not necessarily the same as in which they were sent.

Normally, both actions that form an internal communication are well defined in the specification. However, when an erroneous signal (a signal that is not expected) arrives at a process, the ***SDL***-process performs a "default transition", which is not explicitly defined in the specification. This "default transition" consumes the signal and returns to the same state. In case of communication between the environment and a process, the actions that are performed by the environment are not explicitly defined in the specification as well. The part of the communication that is done by the environment should not be considered in the tests. Therefore, the labeled transition system that represents the state space of the system will only contain the part of the communication that is done by the specified system. Figure 4.4 gives the possible actions is a ***LTS***.

Four possible state transitions can be distinguished. Communication with the environment is handled different from internal communications. In the labeled transition system, only the reachable states are visible. Each state can be reached from an other

[3]The environment of a *SDL*-specification is also modeled as a *SDL* process.

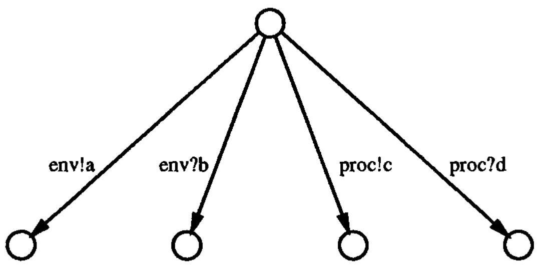


Figure 4.4: Representation of global *SDL*-specified system.

state by a state transition if the action that is tagged to this transition is enabled in de *SDL*-system (i.e. if an implementation of the system might perform that action).

- **env!a**: output-action that puts signal '*a*' on a channel to the environment. The reception of '*a*' by the environment is not visible in the model.
- **env?b**: input-action that reads an input prom the environment. This action is enabled if the input action is encountered in the *SDL*-specification. If no input is available, the process will wait.
- **proc!c**: output-action to an other process (i.e. first part of an internal communication).
- **proc?d**: input-action from an other process. This is the second part of an internal communication, and is *only enabled when the complementary output-action has been executed*. The input action is not necessarily explicitly defined in the *SDL*-specification. The process must, however, be in a *SDL*-state (i.e. waiting for an input).

4.4.3 Asynchronous communication trees

Instead of the *FSM* and *LTS*-graphs a representation that is often used to model concurrent systems are *communication trees*. A communication tree can be constructed from a labeled transition system by "unfolding" the graph i.e. a new branch of the tree is created for every enabled action in a node. Algorithms that generate an asynchronous communication tree from an *SDL*-specification are presented in [DGK91, HB89] and [PRO92]. All algorithms are based on the following three steps:

1. compute the start state;
2. compute the children of the start state;
3. for all possible continuations in a child, repeat step 2.

When the way of building a communication tree is combined with the notion of observable behaviour (see previous section) it is possible to construct an *asynchronous communication tree* (***ACT***). In an ***ACT***, branches from a node exist only for each observable action (page 38) of the system that is enabled in that node. ***ACT***'s are used in most research projects that focus on automatic test generation from ***SDL***-specifications (like the RACE-project "PROVE" and the project "ARTEFACT" at PTT Research).

For cycling processes, a communication tree is not finite. Even when the specification is well defined (in the context if testability) and finite, this way of modeling may lead to infinite models. Various techniques have been proposed to limit the trees. The construction of an ***ACT*** via a certain branch can be stopped if the branch ends in a node that represents a composite state which has already been visited.

4.4.4 SAVE constructs

The SAVE construct can be used to change the sequence in which the massages are taken from the input queue by the process. Typical use of this construct is depicted in figure 4.5. In this situation, the process will wait for the arrival of message **a**. If during that time messages **b** or **c** arrive, they will be "saved" in the input-queue. When the message **a** arrives, it will be taken form the queue (even if it is not the head of the queue) and the process will proceed. If in state "wait input" any other message arrives than {**a**,**b**,**c**}, it will trigger a default transition.

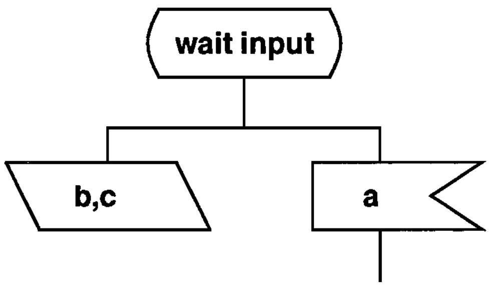


Figure 4.5: Typical use of the signal SAVE construct.

One of the approaches that deals with this signal SAVE construct was presented in [LDB91, LDB93]. The idea of this approach is to extend the description of the model for each case that one of the messages that is to be saved, is received. This way, the SAVE construct can be translated into basic ***SDL***.

4.5 Simple handshake example

The next example will illustrate some problems that arise when using ***SDL*** as the specification language. The example that is used is a primitive transmission system that is able to send a message "m" via a channel.

4.5.1 Description of the example

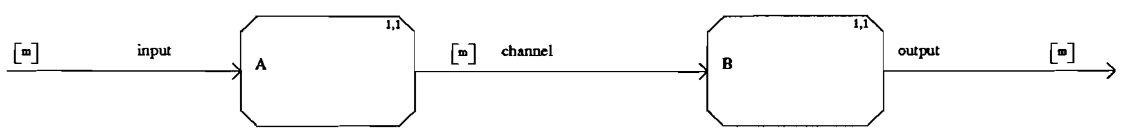


Figure 4.6: Block interaction part of the specification for the "Simple Handshake" system.

The example is a very simple transmission system. Via an input port a message "m" is consumed by process **A**. After that the message is sent via a channel to process **B** which outputs it to the environment.

Figures 4.7 and 4.8 show the two (cycling) processes that perform the service. The processes in an ***SDL*** specification communicate with their environment and other processes via signal buffers. These buffers can hold an infinite amount of messages. This means that an arriving message from the environment can always be consumed and that process **A** can process a new message from its input queue whenever it has *sent* the message to process **B**. The process does not have to wait until process **B** has actually consumed the message.

Figure 4.9 gives the ***LT*** representation of the ***SDL*** specification. The system transfers to an other state when the message is *sent* by **A** and again transfers when process **B** *receives* the message. However an external observer (the tester) could not distinguish between these two moments. Therefore the tester can not distinguish between actions that occur *between* the actual sending and reception of the message and actions that occur *after* the reception of the message by process **B**. Therefore the ***LT***-system can be reduced to an *asynchronous communication tree* (***ACT***). In an ***ACT***, the sending and reception of an internal message is modeled as *one* silent action that takes place at the reception of the message (the notation τ or "int." is used).

An other result of the queues in the communication channels of the processes is the fact that the state space (which contains all states that *can* be reached) is infinite. The graph in figure 4.9 and the graph in figure 4.10 therefore does not end. In the

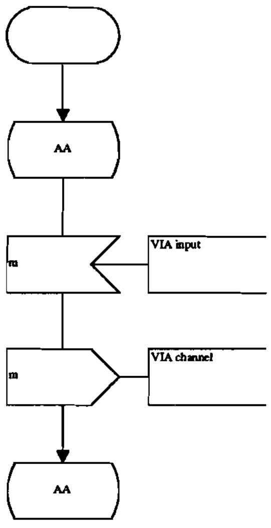


Figure 4.7: Example specification of process **A**.

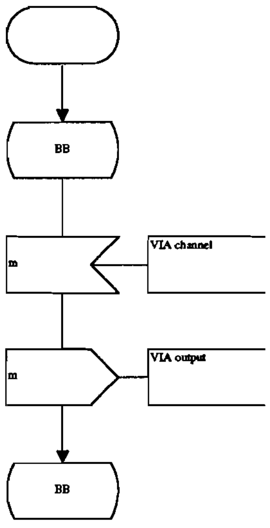


Figure 4.8: Example specification of process **B**.

practical system, the depth of the graph depends on how many inputs are offered to the implementation before the system gets the chance to process the first.

4.5.2 Nondeterminism in the example

In some of the states, the system may act nondeterministically (even though both processes are deterministic). In figure 4.9, such a situation is reached when a message m is put on the channel by process **A**. If an input is presented at the input before the message is accepted by process **B**, two actions may happen concurrently. Either the reception by process **A** or the reception of the pending message by process **B** can happen first. For the tester, there is no way to determine or influence which choice will be made.

After the abstraction from the internal communications that results in the ***ACT*** (figure 4.10), a similar nondeterministic choice can be distinguished. Here, the external observer has no way to determine if the internal action has happened or not.

4.5.3 Infinity of the state space

For the generation of tests, the algorithm should make a search through the state space of the system. The search algorithm (arbitrarily) picks one process as the first and

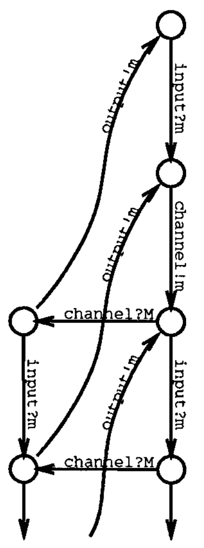


Figure 4.9: Labeled transition model of the handshake example

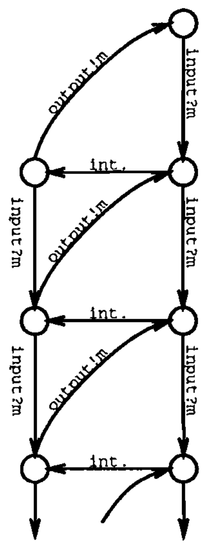


Figure 4.10: Model of the handshake example using an ***ACT***.

executes an enabled action of that process in that state. The environment may send an input signal to the system, before the message has been sent from **A** to **B**. Since the input queue of an ***SDL***-process is unbounded, an infinite number of inputs may be sent to the ***IUT***. Unless an upper bound for the input queue is defined, the search will continue to proceed downwards and will not end (while both processes are finite). If such an upper bound exists, the search continues until that bound is reached, will examine the rest of the behaviour once and will stop.

4.6 Testing SDL-specified systems

With the rising popularity of ***SDL*** as a language to specify communication protocols, the need emerges for automated test generation techniques based on ***SDL***-specifications. However, the large number of features in ***SDL***, such as the SAVE construct that was described in the previous section, are a major difficulty for the existing test generation techniques. This section will describe some of the existing methods that were developed for the automatic generation of tests from ***SDL***-specifications.

The semantics of ***SDL*** ensure that every input that is offered to a system is accepted. Therefore, verdicts of the tester are only passed on the basis of outputs of the system.

4.6.1 ACT-based test methods

A number of test generation methods based on ***SDL***-specifications use the asynchronous communication tree as a starting point. The ***ACT*** is used to represent the composite behaviour of the total system. The internal communications (communications between processes in the ***SDL***-system) is represented by an internal (τ-)action. Of course, the advantage of an ***ACT*** is that it represents the observable behaviour only. This way, a lot of information that can not be accessed by the tester is removed from the model.

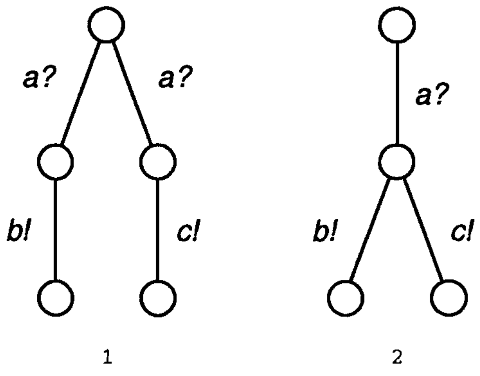


Figure 4.11: Rewriting of nondeterministic behaviour.

From the asynchronous communication tree, test suites can be derived. However, special care must be taken when doing so. An ***ACT*** is not finite in general, nor will the behaviour it describes be deterministic. Some approaches that can be used to limit the size of the generated ***ACT*** will be presented in a next section. The methods that are presented in [PRO92] define a separate test case for each possible outcome of a nondeterministic choice in the system. To do so, the nondeterministic behaviour in the ***ACT*** that is of the form of figure 4.11; part 1, must be rewritten in the the second form (4.11; part 2). When generating tests, the tree will be split into separate test trees at any place where the resulting system makes a nondeterministic choice. When executing the tests, at least one of the test cases must be satisfied by the system. The algorithms that generate the tests from the ***ACT*** will be of the following form:

- eliminate internal transitions from the ***ACT***;
- rewrite internal nondeterminism;
- split the resulting tree into separate tests trees;
- create TTCN test case for each test tree;
- create PASS/INCONCLUSIVE verdict (manually)[4].

[4]The FAIL verdict is often passed via an "otherwise" statement.

In [DGK91], an extended asynchronous communication tree (***XACT***) is used. An ***XACT*** resembles a normal ***ACT*** but can also be used to describe ***SDL***-systems that contain the SAVE-construct. Also, extensions for the inclusion of TIMERS in the specification are presented. The test generation techniques that are used here are adapted from ***FSM***-based test generation techniques that were presented in chapter 3. The adaptions account for the possible nondeterminism in the resulting finite state machines. To use these methods, that generate tests with much higher fault-coverages as the previous one, the infiniteness of the ***ACT***-model must be eliminated. In the section about behavioral constraints, later in this section, some possible solutions will be presented. The structure of the test generation methods is as follows:

- create asynchronous communication tree from ***SDL***-specification;
- reduce ***ACT*** with assumption that external inputs are only generated when the system is in a stable state;
- map the reduced ***ACT*** to a state machine;
- reduce the number of states, such that the state machine is finite;
- derive test suites with the help of algorithms from the PTT Conformance Kit.

4.6.2 SDL-machines

An other known approach to test generation from ***SDL***-specifications is presented by v. Bochmann et al. in [LDB91, LDB93]. These papers are based on the idea that ***SDL***-specifications that contain the SAVE construct can be rewritten to specifications without the SAVE construct, while preserving the same external behaviour. These specifications can be translate to finite state machines, that can be used as a basis for automatic test generation.

The first step in their approach is the construction of an "SDL-machine" from the process specification. Such an SDL-machine is a finite state machine *with an input queue and SAVE's*. The model is generated by eliminating parameters and variables that do not influence the dynamic behaviour directly. The variables that control branches in a decision part of a process specification are preserved in the model by defining a separate input message for each value or range of values in a data field of an input that may cause a different branch in a decision block. Problems are the non-finiteness of the SDL-machine and the explosion of the number of states.

After the generation of the SDL-machines, which possibly contain SAVE-blocks, the machines are transformed to finite state machines that do not contain SAVE's. In most cases this can be done by an "equivalent transformation", which preserves the input/output relation of the original SDL-specification. On some cases an "approximately equivalent transformation" must be used. After the generation of ***FSM***'s, The well known ***FSM***-based test generation techniques can be used to create a set of tests.

To only problem that is not addressed in these papers is the problem of generating a single process specification for the total ***SDL***-system. While in general a specification that was written in ***SDL*** is built from multiple communicating processes, v. Bochmann et al. assume that a single-process description is available. The ***SDL***-specifications that are the subject of this report are assumed to contain more than one process.

4.6.3 Behavioral constraints

As for ***FSM***-based automatic test generation techniques, one of the major problems for test generation based on ***SDL*** is the explosion of the state space when creating the internal representation. The ***ACT***'s and ***FSM***'s, that are used to represent the state space, therefore, will grow intolerably large. In the RACE project PROVE, much attention is paid to techniques that limit the size of the ***ACT***-representation. **Behavioral constraints** are used for this purpose. These constraints restrict the behaviour to only certain aspect of the behaviour, and suppress other parts of the ***ACT*** that are not related to these aspects. This way, tests can be generated that cover only a part of the total behaviour. If necessary, the procedure can be repeated for other parts of the behaviour. The metrics that are used to control the behaviour of the total system include:

- definition of a maximum trace depth;
- definition of a reasonable environment;
- definition of a maximum number of occurrences of a signal;

maximum trace depth

The definition of a maximum trace depth is a simple heuristic that can be used to restrict the behaviour to a certain number of steps that can be taken from the starting state. If the constraint is defined independently from the specification, the chance exists that important behaviour is discarded. However, the definition of such an heuristic value on the basis of the specification is a very difficult task. Often a "natural" bound to the number of steps in a trace is presented by the capacity of the machines that compute the traces.

reasonable environment

In a normal situation, the construction of a composite state space includes all possible behaviour that can be executed by the system. The reaction of the system to each possible behaviour of the environment is specified. Often, this behaviour includes non reasonable behaviour (i.e. it forces the system to do things, it was not designed for). Since not all behaviour can be tested, it seems appropriate to start with the "reasonable" behaviour. In [PRO92], three forms of a reasonable environment are presented:

- only one message will be queued per ***PCO***;
- the environment will only send a message if *all* queues in the system are empty;
- signals that will be consumed with an implicit transition will not be sent.

The first environment implements the heuristic: "*wait until the last-sent message is consumed*". The second environment is based on a resemblant heuristic. However, in this case the tester will wait until no possible transition can be performed by the system spontaneously. The second environment puts a larger restriction on the possible behaviour than the first.

The third environment restricts the behaviour of the tester to only the valid behaviour. In practice, however, it will be difficult to determine on forehand if a message will be discarded or not, due to the nondeterminism in the system. In combination with one of the first two reasonable environments, it is easier to anticipate which messages will cause an implicit transition.

maximum number of occurrences of a signal

This metric can be used to exclude large (or even infinite) loops in the system. Both the depth as the breadth of the state graph can be reduced this way. For each signal that is exchanged with the environment, the number of occurrences is counted. If the this number exceeds some heuristic value, no new signals will be sent to the system.

All the methods that are presented here can be used to limit the size of the internal representation of the system. Still it appears that for large systems, these methods do not suffice. In the next chapters, some methods will be presented that have been developed in the field of protocol verification.

5 Partial order reduction strategies in protocol verification

5.1 Protocol verification

One of the research areas in which much attention is paid to the state space explosion problem is *protocol verification*. As shown in figure 1.1, the verification of a protocol takes place during the specification phase. This section will be based on the assumption that the specification of the system is given as a set of communicating finite state machines. The composite state space is the set of all states the system can occupy, i.e. the Cartesian product of the state spaces of all separate processes. In practical systems only a fraction of this composite state space is *reachable*. However, the fraction of the composite state space that is reachable is still very large for practical systems due to the *state space explosion*. One of the key subjects in protocol verification is the question of how to fully explore the composite state space within the practical limits of time and memory.

After the specification is made, and before the next steps in the design cycle (e.g. implementation or test derivation) can be taken, it must be ensured that the specification is logically consistent. This means that the specification is checked for various *design errors* that are independent from the actual function of the system. Logical errors that may appear in a specification are:

- **incompleteness**: a message that was sent by one process cannot be received by at least one process, or a process expects a message that cannot be sent.
- **use of uninitialized data**
- **system deadlock**: a **deadlock** in a system composed of a number of concurrent processes, is defined as a reachable state of the system in which all processes are blocked [God93]. The property of a system that a deadlock could not appear is called the *safety property*.
- **livelock**: a **livelock** or **non-progress cycle** is a situation in which no progress

toward providing the desired services takes place [ULS90]. The absence of livelocks is called the *liveness property* of a system.

- **buffer overrun**: In systems that have upper bounds for queue lengths this would mean the loss of a message. In many practical methods, an upper bound is assumed to ensure that the algorithms end. A **buffer overrun**, in this context, means that the assumptions about these bounds must be revised.
- **unspecified receptions**: the reception of a message that was not expected by the process in this state.
- **race conditions**: **critical races** between messages or timer. Usually these races denote (implicit) assumptions about the sequence of messages.
- **unreachable code**: parts of the specification that are *never* reached during the execution of the protocol usually denote an error.
- **violation of *user specified* correctness assertions**: the user may add extra assertions, like **invariants** or **temporal formulae**, to check the consistency of the specification. Examples are:
  - invariants (expressions that must always be TRUE);
  - fairness (IF expr_a EVENTUALLY expr_b);
  - precedence (expr_a IMPLIES expr_b UNTIL expr_c).

The first two items can be verified during compilation of the code or a semantics check of the specification. All other items can only be checked during exhaustive *symbolic execution* of all possible state transitions. In this symbolic execution, a search through the composite state space is performed that visits every reachable state and explores every possible state transition. In contrast with conformance testing, there is no controllability or observability problem in protocol verification. During the execution, all details about state transitions in the specified system are known. This means that, if the full composite state space can be constructed from all separate processes, every state can be examined. The best-known algorithm to perform such exhaustive symbolic execution is **reachability analysis**. The search through the composite state space of the system resembles the search that must be performed when automatically generating conformance tests.

The composite state space is built by executing every enabled transition in every state that is reachable from the initial state. The explosion of the composite state space during this search is one of the major problems that limit the applicability of existing verification methods. Since this problem also exists for automatic test generation methods, solutions from the field of protocol verification could also be applicable to test generation. In this chapter a number of state space reduction techniques are presented that have been presented for protocol verification.

5.1.1 Reachability analysis

There are many algorithms that perform reachability analysis. They differ in the resources they need to be executed, the fault coverage, the number of states that can be explored, the extent in which the analysis can be controlled, etc. For a class of medium-sized protocols (up to 10^5 states, according to [Hol91b]), a full (exhaustive) reachability analysis can be carried out. The simplest form of such an algorithm is a **depth-first search** (figure 5.1). In the algorithm, two sets are defined: $\mathcal{S}$, the **search stack** this is the set of states that are being explored at the moment. The set $\mathcal{D}$ is the set of states that have been fully explored (i.e. all transitions starting from them have been explored). When the algorithm ends, $\mathcal{D}$ contains all *reachable* states.

```
depth_first_search

$\mathcal{S}$ := {initial_state};                      /* search stack */
$\mathcal{D}$ := $\emptyset$;                                  /* visited states */

DO  WHILE $\mathcal{S} \neq \emptyset$
    $s$ := last element of $\mathcal{S}$;
    IF error($s$)=TRUE
          report("error!");                   /* an error was found */
    ELSE  DO for each successor state $s'$ of $s$
               IF ($s' \notin \mathcal{S}$ AND $s' \notin \mathcal{D}$)
                   add $s'$ to $\mathcal{S}$;
               FI;
          OD;
    FI;
    delete $s$ from $\mathcal{S}$;
    add $s$ to $\mathcal{D}$;
OD
```

Figure 5.1: Depth-first search algorithm.

The function "error(s)" should be adapted to the sort of errors from the list above that have to be detected. In the simplest form, the function is TRUE (i.e. reports an error) when there is no way to exit from the current state (**deadlock**). More elaborate implementations of the algorithm could check sequences of state transitions to temporal formulae or invariants.

In the depth-first search algorithm, new states are selected from the "end" of the stack $\mathcal{S}$. If the *first* element of $\mathcal{S}$ is selected, the search changes to a **breadth-first search**. A breadth-first search has the advantage that the shortest sequences leading to an error are found first. Usually, however, a depth-first search is used. The advantage of a depth-first search is that the search stack is smaller and that a **back-trace** (a sequence of state transitions that leads to the specific error) is easily created.

5.1.2 State space explosion

An exhaustive search can only be used to verify medium-sized specifications. During the search, the algorithm must keep track of all states that have been visited, in order to detect loops. These states are stored in the sets $\mathcal{S}$ and $\mathcal{D}$. Due to the state space explosion, these sets will become too large to be stored in memory (considering the large amount of data that must be stored each state). In the case of very large protocol specifications, the designer has the choice between storing the information in memory, which is fast, but is limited by the size of the memory. If the amount of data that must be stored is too large, the information may be stored on disk. This solution, however, reduces the speed of the search by several factors. Holzmann indicates that systems with about 10^5 reachable states is the maximum which can be handled by exhaustive search with the current technology [Hol91c]. New methods are needed to analyze very large protocol specifications. One way to do this is by improving the way, the state information is stored. This is implemented by Holzmann in a method called **supertrace**. With this method about $5 \cdot 10^8$ states can be analyzed. An other, more fundamental approach is to create smaller state spaces that can be explored by computers. However, it is important that design errors in the original protocol still can be found in the reduced state space.

5.2 Reduction methods using heuristics

An overview of the possible reduction strategies is presented in [LCL87]. There are three categories which are applied during different phases of the design process:

- strategies that improve the specifications:
  - restrict the number of parameters;
  - limit the queue lengths;
  - define upper and lower bounds for the variables and parameters that are used.
- strategies that divide the specification in smaller parts:
  - partition the specification in multiple **phases** of the protocol;
  - make a projection of each function (group) of the protocol.
- strategies that guide the partial search:
  - explore "dangerous" situations first;
  - cover the largest possible part.

5.2.1 Improving the specification

Much attention is paid to the first category of heuristics by the standardization institutes. These methods are applied at an early stage in the protocol design. Most verification methods that have been presented, assume that some properties from this list have already been carried through. Much improvement can be gained from these methods. These aspects of the specification are the same that determine the "testability" of a system (see section 2.4).

5.2.2 Splitting the specification

The division of the specification is something that was also described in the previous chapter (section 4.6.3). These strategies are an important aid for the verification of very large specifications. The implementation of these methods, however, is very difficult and no general applicable method has been found yet. Because there is no limited controllability when verifying a protocol, the splitting of a protocol can be done more rigorously than when testing the implementation. The protocol could be split in different parts, that each represent a different phase of the protocol, such as call set-up, data exchange and disconnection phase. A second method that is mentioned is the projection of different functions on a new smaller protocol. In general such a projection will be very difficult on the level of formal specifications. Especially in complex protocols, the different functions are usually hard to isolate.

5.2.3 Partial searches

Methods have to be found to explore as large parts of the composite state space as possible. Besides that, much effort is done to direct the searches in such a way that the largest part of *possible* errors is verified. These methods are known as **partial state explorations** (figure 5.2). Most early partial search methods concentrate on finding effective heuristics to decide which state transitions have to be explored, and which have not. Heuristics, however, do not guarantee that all errors in the specification will be found. Lately, new methods have been presented that reduces the state space, and preserve the completeness of the specification. Since is seems reasonable to expect that these methods to reduce the state space for verification are applicable to the automatic generation of conformance tests, a survey of these methods is presented.

In practical cases, a search through the state space can not be complete due to the state space explosion. Partial searches are meant to control the search through the state space of a system in such a way that as much errors as possible are found. Even though not all states are visited, a major part of the errors in the specifications can be found since most errors will occur more than once in the behaviour of a system. Next some important methodologies will be described.

```
partial_search

/* initialization */

DO  WHILE $\mathcal{S} \neq \emptyset$
    $s$ := last element of $\mathcal{S}$;
    IF error($s$)=TRUE
         report("error!");                          /* an error was found */
    ELSE  DO for some successor state $s'$ of $s$
              IF ($s' \notin \mathcal{S}$ AND $s' \notin \mathcal{D}$)
                  add $s'$ to $\mathcal{S}$;
              FI;
          OD;
    FI;
    delete $s$ from $\mathcal{S}$;
    add $s$ to $\mathcal{D}$;
OD
```

Figure 5.2: Partial search algorithm.

i **depth-bounds**:
The most simple method to limit the number of states is to restrict the length of the execution sequences. This is a fast, but rather crude option. It can be combined with other methods in order to guarantee termination of the algorithm.

ii **scatter search**:
During a **scatter search**, "dangerous" situations are visited first. For instance, potential deadlock states can be recognized by the absence of any pending messages. Therefore, in a scatter search, receive operations are favored over send operations. Deadlock errors can be found quickly this way.

iii **guided search**:
A more general case of the scatter search is the **guided search**. Instead of favoring one sort of behaviour over an other to provoke errors, a cost function guides the search. The definition of this cost function is very difficult and not much experience is available.

iv **probabilistic search**:
Probabilistic search favors the states that are most likely to occur in the real system. In this way, information about the environment (e.g. the probability of certain message sequences) can be taken into account.

v **random search**:
Based on the idea that an error in a specification is very likely to appear many times in the state space, a **random search** visits states from as many different parts of the specification as is possible. An advantage is that the amount of memory and CPU time

that is used, can be controlled easily. In a random search, all parts of the specification are visited. Because of its simpleness and good fault coverage, this method is a commonly used heuristic.

vi **fair and maximum progress search**:
In the composite state space of a system, the same action will occur many times. This is caused by the linearization of the concurrent processes from which the state space in built. By achieving a certain amount of progress through the state space of the separate processes, these algorithms try to skip a part of the interleavings of independent actions.

5.3 Reduction methods using partial orders

This section will describe the partial-order based reduction methods that are presented by Holzmann et al. in [HGP92]. This group of reduction strategies are fundamentally different from the heuristics mentioned in the previous section since they preserve the completeness of the specification. Since there is only a partial ordering between actions in the system, **equivalence relations** can be defined for sequences of actions. If two paths through the composite state space are *equivalent*, it is sufficient to examine only one of these paths. The problem of these techniques is finding an equivalence relation that preserves the distinction between *correct* and *faulty* behaviour. Equivalence relations can be ordered from *weak* to *strong*. Weak equivalence relations group large groups of behaviour, yielding large reductions. Strong equivalence, in return, yields less reduction but is better in preserving the fault coverage of the verification.

5.3.1 Reduced search method

The reduced search algorithm is based on the notion that an action that does not interact with other processes, is always independent of all other processes. If a process is about to execute an independent action (called a **local** action), it is not important if an action of another process has been executed or not. The process has no means to determine what other processes are doing at that moment, nor can any other process "see" if the action was executed or not. Therefore, all interleavings of the local action with actions of other processes hold the same information about the system.

Similarly to the "normal" reachability analysis, a depth first search is performed by the algorithm. The difference is that for the reduced search algorithm, some state transitions will be blocked during the search (i.e. the search is not continued via this path). Suppose the graph in figure 5.3 is a part of a composite state graph of the system. The labels "a" and "b" denote actions from two different processes. If a and b are *independent*, the (sub)traces $< a : b >$ and $< b : a >$ hold the same information about the system's behaviour. This means that it should be possible to find the same errors by only examining one of the traces. In appendix A, more details are given about the algorithm that implements the reduced search method.

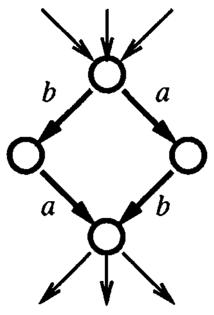


Figure 5.3: Part of a composite state graph.

Holzmann proves that the following correctness properties still can be established with the algorithm (i.e. the equivalence *preserves* the following properties):

A. Each process performs a valid computation, without violating any local assertions, and without cycling or hanging (safety properties).

B. All processes together proceed from a known composite initial state to a known composite final state.

To test the presented algorithms, a number of small protocols were evaluated. In the paper the number of visited states and executed transitions are given as well as the run time and memory requirements. The following test protocols were used:

1. 5 independent processes, each traversing 10 local states, no cycles
2. 5 independent processes, each cycling through 10 local states
3. 5 completely dependent processes, each traversing 10 states, without cycles

From the first test protocol, large reductions can be expected. Since all processes are completely independent, only one path that executes all actions once is sufficient to verify the composite system. The second test protocol forces the search algorithm to visit every reachable state. The third test protocol is used to give an indication of the overhead that is induced by the algorithm. Figure 5.4 shows a comparison of the algorithm with a standard (Dept-first) search for the required amounts of CPU-time and memory. It is clear that behaviour of the first category in the list can be reduced best. In this case only a fraction of all reachable states is actually visited. In the verification of the second test protocol, all states have to be visited because of the cycles. Still a reduction is achieved since not all transitions have to be explored. The overhead of the algorithm manifests itself mainly through the increased amount of time that is necessary.

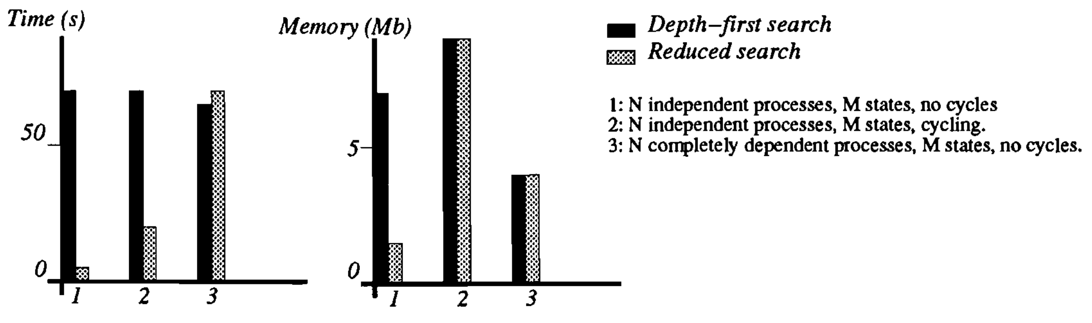


Figure 5.4: Results of the reduced search algorithm.

In the paper by Holzmann et al. some results with more realistic protocols are presented. A typical reduction of ±25% is achieved with the reduced search algorithm. Searching for local transitions in not the only way to find independent transitions. More elaborate methods are presented in the next two sections.

5.3.2 Sleep Sets

The theory of sleep sets was presented a few years ago [God90, God93] as a means to avoid the part of the combinatorial explosion due to the modeling of concurrency by interleavings. In [HGP92] the name *conflict set method* is used. In this report, the theory as it was presented by Godefroid will be used as the starting point. This section will give a short impression of the method. The algorithm is presented in appendix B.

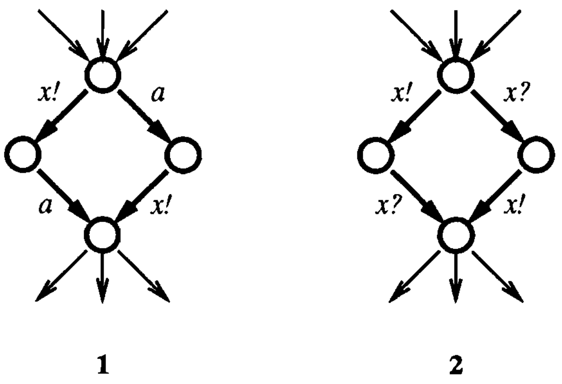


Figure 5.5: Part out of the composite state graph. "x!" and "x?" denote actions that use a variable or buffer called "x".

Suppose the two graphs in figure 5.5 represent a part of the composite space of a system. The labels "$x!$" and "$x?$" denote actions that use a variable or buffer called "x". A label "a" denotes a *local* action. Such a local action is independent of all actions of all other processes, actions that use common variables or buffers (like $x!$ and $x?$) are dependent.

The reduction in the *sleep set* method is based on the assumption that after a statement x is executed during the depth-first search, it should only be repeated in that search if a *dependent* action can be shuffled ahead of x. So, in contrast with the reduced search method, the sleep set method looks at the actions of a process instead of actions in the composite state space. To execute the reduction algorithm, the state space model is extended with a *sleep set* for every action. A sleep set can contain *conflict tags*. This sleep set blocks the action after it has been executed once by placing one or more conflict tags in it. The block is lifted when an action is executed that *conflicts* with one of the tags in the set (i.e. when a dependent action is executed).

In the example in figure 5.5, the search algorithm will "arrive" at the upper node. Assume that the leftmost actions are executed first. In the first graph, the sequence $< x!{:}a \ldots >$ will be executed first. When all behaviour is examined, the search will "**back-up**" to the first node. At this point, the algorithm fills the sleep set belonging to action '$x!$' and continues with the a-action. Since this action is not dependent of $x!$, the block on $x!$ is not lifted, and the search will halt at this point and back-up. The second appearance of the $x!$-action is not executed.

The same holds for graph 2 until the search backs up for the first time. Now $x!$ will be block as in graph 1. The execution of $x?$, however, conflicts with $x!$ (i.e. both actions are dependent) so the block is lifted. Both sequences $< x!{:}x? \ldots >$ and $< x?{:}x! \ldots >$ are examined. This is necessary since the results of these two traces could be different. Imagine, for example, that $x?$ reads a message from a queue x; In the first trace, the queue will not be empty (since the read-operation is preceded with a write-operation). In the second trace, however, queue x can be empty and the system will be forced into a deadlock state by the execution of $x?$.

In [HGP92], some results are presented. The same sample protocols as in the previous section are used. The results, that are drawn in figure 5.6, show that a much larger overhead is introduced by the algorithm. The results for the sleep set method are not as promising as they were for the reduced search method. However, the reduction of the sleep set method seems to have effect on different kinds of protocols. In the case of the protocol that consists of cycling, independent processes, the memory usage is smaller for the sleep set method. Holzmann shows that the efficacy of the reduction methods is improved when both algorithms are used in combination with each other.

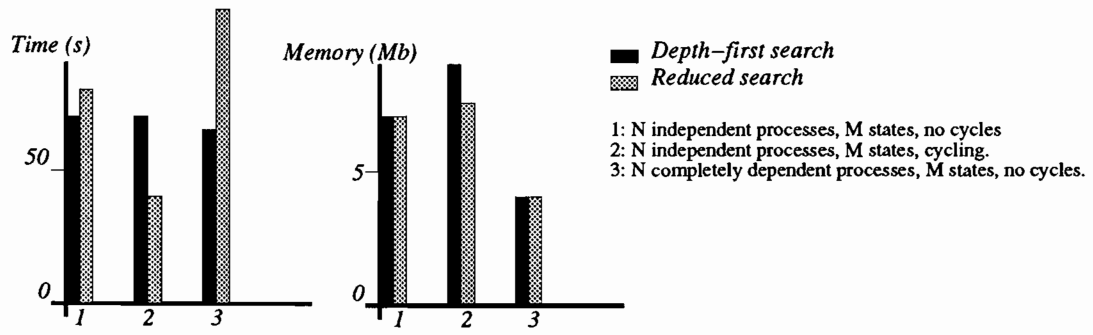


Figure 5.6: Results of the *sleep set*-algorithm.

5.3.3 Stubborn Sets

The last algorithm that is presented by Holzmann et al. is the *stubborn set* algorithm[1]. Important contributions to the theory for this method have been presented by Valmari [Val90, VT91]. As in the sleep set method, this method uses dependencies between different actions of processes. Only in this algorithm, information is used about which processes use which variables. The reduction is achieved by stating that all transitions of a process **P**, which do not use a certain variable q, are independent of transactions that *do* use this variable. The method was presented as a means for program verification. Therefore, the communication between processes is done via variables. In the context of communication protocols, one should read "queues" instead of "variables".

Assume a process **A**, which uses the variables $\{x, p, q\}$, a process **B**, which uses the variables $\{y, p, z\}$ and a process **C**, using $\{y, q\}$ (figure 5.7). In state "s", both process **B** and **C** are about to execute an action that accesses (e.g. reads from or writes to) variable y. The algorithm will only select the actions $B?y$ and $C!y$ for the continuation of the search (this is where the name *stubborn set* comes from; this set contains all transitions for a specific node that must be explored).

No results are presented for this method. Early experiments have already proved that the parts of the composite state space that are blocked are different for all presented methods. As for the sleep set method, the overhead that is induced by the algorithm will be larger than for the reduced search method. Further investigations are necessary to determine whether this algorithm can be combined with other reduction methods.

[1]A method which uses the same starting points is *Overman's Method*. However, more information is available about the stubborn set method. Only stubborn sets will be presented here.

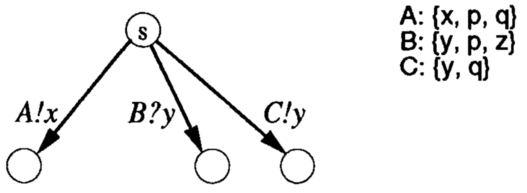


Figure 5.7: Sample part of a composite state graph. Each transition belongs to a different process.

6 Application of the partial order reduction methods

Holzmann's Reduced Search method [HGP92] can be used to identify a set of traces in a labeled transition system. Each trace is a representative of an equivalence class of traces in the graph. The classes have multiple members since there is only a partial ordering of the actions in concurrent systems. Together, the equivalence classes cover the behaviour of the specified system[1]. Using the structure and semantics of a ***SDL***-specification, the equivalence classes may be used to improve the test generation methods by noting that not all representatives of an equivalence class have to be tested. This observation can be used to reduce the number of tests that are generated and the resources that are needed to create the test sequences.

The major problem is the limited controllability of the system. Only few of the states in the ***LTS*** can actually be forced upon the system in a deterministic way. The existing test generation techniques are based on the assumption that sequences of actions can be forced in the implementation under test. For the application of the partial-order reduction methods, a distinction must be made between controllable and non-controllable behaviour in the system. In this chapter, a test generation method is proposed that pushes the system from one stable state to the next. The reduction methods that are presented by Holzmann et al. can be applied to this method.

6.1 Controllability of traces

In order to be able to apply the reduction methods from the previous chapter, an internal representation of the system will have to be generated from the ***SDL***-specification. In this chapter a labeled transition model is used as the internal representation. This model also contains the systems internal actions, which may not be controllable from outside the system. In appendix D, an example is given of the application of the partial order methods from the previous chapter.

[1]The set of traces may contain some redundancy in a sense that two traces may belong to the same equivalence class.

6.1.1 The LTS model

There are different models for the representation of the state space of a protocol. The models differ in the amount of information that is available about the internal situation of the system and the definition of the state transitions. For instance, a state transition in an ***ACT***-representation can only happen via an observable action, while the same transition may involve many actions and multiple states in a labeled transition system. The labeled transition model holds the dynamic behaviour of the system. In general, no data-dependencies are defined.

The reduction methods that are used rely on a partial ordering of actions in a system. When some actions are not visible in the model, this may cause the blocking of state transitions that should not be blocked. Therefore a model will be used that contains much information about the system. In this chapter, the internal representation of the composite state space that is used is the labeled transition system, as defined in section 4.4.2. In this model, five different actions can be distinguished:

- input actions from the environment;
- output actions to the environment;
- input actions from an other process;
- output actions to an other process;
- τ-actions.

The labeled transition model that is used, contains both observable and non-observable actions. The τ-action is used to model behaviour that is unknown in the model (such as ***SDL***-tasks or decisions based on unknown parameters). Note that the internal input action is only enabled (i.e. present in the model), if the relevant internal output has been sent. Like in the ***ACT***-model, a state contains information of each process and the current values of buffers and variables. The reachable states in the system can be grouped in two sets:

- **stable states**: these are states in which no state transition is possible, unless an input from the environment is provided (and all timers are expired).
- **non-stable states**: these are states in which the system may transfer to a new state without any stimulus from the environment.

6.1.2 fairness

In order to deal with non-stable states some properties of the system have to be assumed. The first assumption is that the system is *fair*. The definition of fairness that is used here is rather strict and only applicable to non-stable states. In the literature, more general definitions can be found.

Definition 7 (Fairness-1) *If the system is in a certain state in which it can show some behaviour without being triggered by the environment, it will do so eventually.*

This notion of fairness can easily be verified in practical systems: The fact that an action is enabled, denotes that the system should be able to perform the action. If the system refuses to execute the action within finite time, the system is malfunctioning. This notion of fairness is, however, not sufficient in the rest of this chapter. Therefore, an extended definition of fairness will be adopted here. One of the assumptions that was made about the specifications was that a thorough verification of the specification has taken place. In this chapter it is assumed that the specification contains no livelocks. Now assume a system that performs some behaviour A:

$$A = B + \mathcal{X} : A$$

In theory, this system could repeat the actions $< \mathcal{X} : \mathcal{X} \ldots >$ infinitely. The kind of fairness that is assumed hers is that, within a finite amount of time, the behaviour specified by B will be executed too. This notion of fairness is closely related to the one given in [Koo91].

Definition 8 (Fairness-2) *If the system is in a loop from which it might break via a nondeterministic choice without any stimulus from the environment, it will do so eventually.*

A system is only fair according to definition 8, if no livelocks exist in the system and if the contents of the data fields in the messages are properly chosen. In ***SDL***, the delivery of a signal, once it is sent, is guaranteed. The assumption that the system under consideration is fair according to definition 8, has important consequences. If it is assumed that there is at least one reachable, stable state in the system, it is guaranteed that from any reachable state, the system will eventually reach a stable state. Since the selection of the next action in a non-stable state can be nondeterministic, the choice between different traces that are executed by the system may be nondeterministic as well.

6.1.3 controllability

Paths through the state space that are computed with a generic test generation method, will contain stable and non-stable states in general. If the paths are to be used for test generation, the tester must be able to force the system into each node in the path. Since the tester has only access to the outside of the system, this may be very difficult. This will be demonstrated with the help of a small example (figure 6.1).

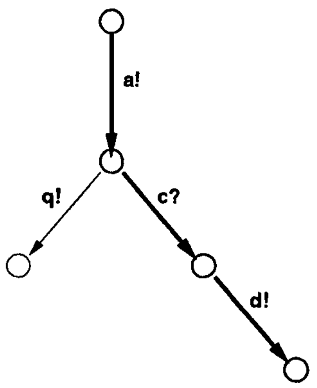


Figure 6.1: Sample part of a labeled transition system (only communication with the environment).

The figure shows a small part of the labeled transition system of a system. Five reachable states are drawn. An exclamation mark denotes an output to the environment, a question mark an input from the environment. Suppose the tester would want to test the trace $< a!, c?, d! >$ from the up-most state. After the execution of the a-output, the system reaches a non-stable state. The c-input *can* be executed. However, the system may also transfer to another state via a q-output. The choice between 'q!' and 'c?' can not be guided from outside the system. Those choices generally are determined by the order in which signals arrive at their destination. Since most of the time it is not possible to make any assumption about the time, signals need to reach their destination within the semantics of ***SDL***, the tester can not force the 'c¿ action in this case. Therefore, the trace can not be tested directly.

6.2 Stable traces

In this section, the non-stable states in the state space will be investigated. If a path through the composite state graph contains non-stable states, it is not always possible to force the sequence of actions in the implementation. Stable traces will be introduced as an aid that will be used in later sections.

6.2.1 Non-stable states

In this section it will be assumed that processes do not "die" before they have reached an end-state. Different kinds of non-stable states can be distinguished:

- **Non-stable states with only one input or output-action enabled.** Since fairness was assumed it is guaranteed that enabled output-actions will get executed eventually. The only problem in the context of conformance testing is the question of how long the tester has to wait until it is legitimate to assume that no output is or will be generated. If a single internal input is enabled this means that the message is already sent. In this case, the action will be executed upon arrival of the signal. If the enabled input action expects an input from the environment, the state is stable.
- **Non-stable states with more than one output enabled.** In such a state the choice between two outputs (i.e. two traces) will be made nondeterministically. The tester has no way of knowing which choice will be made and should therefore expect all possible continuations of the trace.
- **Non-stable states with more than one input enabled.** If all inputs come from the environment, the state is stable. If one or more of the input-actions consumes an input signal from an internal communication, the tester may no longer be able to control the future behaviour of the system deterministically.
- **Non-stable states with input and output-actions.** All enabled output-actions and enabled internal input actions can be executed. If a subset of the processes expect an input from the environment, the tester can not force these actions in a deterministic way.

The non-stable states were defined as *reachable states* in which the tester can not control the choice of the next action that is executed. Using the difference between stable and non-stable states, it is possible to define a *stable trace*:

Definition 9 (Stable Trace) *A stable trace is a legal sequence of actions that begins and ends in a stable state.*

Behaviour that leads to non-stable states with multiple outputs or both input and output-actions can not be specified in a single *SDL*-process directly (within the semantics of *SDL*'88). Non of the separate processes could show behaviour that allows two different outputs in the same state since this would imply nondeterministic behaviour which is not allowed in the specification of a process in *SDL*. Also, a state that allows both input and output actions is not allowed in a single process. The semantics of *SDL* do not allow such a state. Therefore those kinds of non-stable states could only result from the combination of actions of parallel processes in the system[2].

In *SDL*'88, a state with multiple enabled inputs can not be defined in a single *SDL*-process since this would imply that the choice between those input-actions is nondeterministic. However, the choice is completely defined by the order in which

[2]In the 1992 version of *SDL*, the ANY-construct can be used to define processes that can choose between two actions nondeterministically.

the signals arrive at the input port of the process (there is only one input queue per process). One way of introducing states with multiple inputs is via multiple parallel processes that each expect an input from another process. There is a second way to introduce non-stable states with multiple inputs in the state space. When multiple input signals are enabled at a process that arrive via different channels (figure 6.3), the order in which those signals arrive is nondeterministic. Therefore, the choice between the two actions in the ***LTS*** will be nondeterministic. To express this kind of nondeterminism, states with multiple input-actions should exist in the state space.

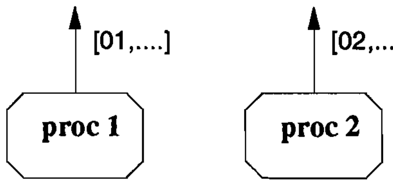


Figure 6.2: Two parallel processes that generate an output concurrently.

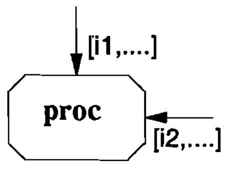


Figure 6.3: Single process with two sequential outputs.

A simple cases of a non-stable state that results from concurrency is a state in which exactly two outputs are enabled simultaneously. A possible situation that leads to a state with two outputs is depicted in figure 6.2. The non-stable state is formed by two processes that both can send an output (O_1 and O_2). In the non-stable state in the global state space, each of the outputs may be send first. In both cases (figure 6.2 and 6.3), both enabled actions will be executed eventually. This means that a finite number of actions after the execution of one of the two actions, a state will be reached in which both output actions have been executed. In the case of multiple outputs, this is the result of the assumption that the system is fair. In the case of multiple inputs, all inputs must be executed because the signals have already been sent (and their arrival is guaranteed).

In the case that a non-stable state enables both input and output-actions, the situation is more complicated. The first case is the situation in which the actions belong to different processes (figure 6.4). The first possibility is the case that all enabled input actions consume an input from the environment. In this case (one of the) the remaining outputs will be executed if the tester waits long enough. The input actions that were enabled will still be enabled. The result of this reasoning is that eventually, all output actions will be executed. The input actions will remain enabled. If the system reaches a stable state without an input from the environment[3], at least the originally enabled input actions from the environment will be enabled.

When input-actions are enabled that expect an input from an other process, this means that the expected signal must already have been sent. Since the arrival of a

[3]An example of a system that does not reach a stable state can easily be found. Practical protocols, however, will reach a stable state eventually.

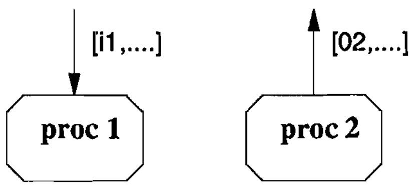


Figure 6.4: Two parallel processes that are able to generate an output and consume an input signal concurrently.

message that is sent is guaranteed, the enabled inputs will be executed before the system reaches the next stable state and without any intervention from the tester. Each process will perform its behaviour, independently of the actual order of execution each separate action (though it may lead to different behaviour in the future).

The previously described situation, in which multiple inputs are enabled of the same process, could pose a problem. This situation indicates that multiple signals have been sent to the receiving process and that they may arrive in "arbitrary" order. When a state with multiple inputs in the global state space represents a choice of a single process, the future behaviour of the system depends on the specific choice, and there is no guarantee that the other input-signals will be expected by the process. Different from the previous situations, the process will show different behaviour, depending on the order of execution of the enabled actions

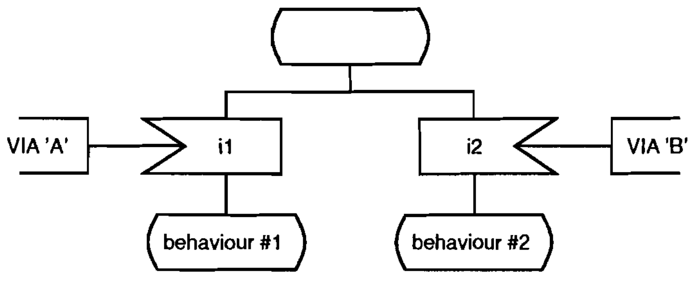


Figure 6.5: Process that is able to receive signals via two channels, 'A' and 'B'.

A situation in which a single process chooses between different behaviour on the basis of the sequence of arrival of signals implies that the system behaves nondeterministically. In the implementation this would yield critical races between signals. It is not likely that such behaviour is to be specified and therefore it seems acceptable to assume that such behaviour will not be specified.

resumé

In *SDL*'88, it is not possible to describe a process that behaves nondeterministically. Yet the composite system will contain states in which the system can be choose nondeterministically between multiple actions. This nondeterminism has three possible causes.

- **concurrency**: both actions can be executed by a different process. The behaviour of the composite system does not depend on the actual sequence in which the actions are executed.
- **critical races**: if both actions can be executed by the same process, the future behaviour depends on the order of arrival of the messages. This order can not be determined if both messages arrive via different channels.
- **through constructs in *SDL*'92**: In *SDL*'92, a number of constructs are available to describe a process makes nondeterministic choices. If a nondeterministic choice is made between one of two actions, this could imply that the second action of never executed.

According to the definition, a stable trace starts in a stable state. The only actions that are enabled in a stable trace are external inputs. In this report, a stable trace that starts with an input i will be denoted as $t(i)$.

6.2.2 Existence of stable traces

The first assumption that is made is that the system will reach a stable state after start-up. So there must be at least one stable state in the system. The reaction of the system after an input from the environment, can be divided into phases as in figure 6.6.

Suppose that the first state that is encountered is the up-most state in the figure. The three actions that are enabled in this state can be output-actions or internal inputs, but each action belongs to a different process. If there are no livelocks in the system, the fairness assumption ensures that all three action will get executed eventually (this is the second phase). There will be at least one state in the labeled transition system (and possibly more) in which all three actions have been executed. These states are not necessarily stable, since new output actions or internal communications may have been enabled. However, the system will reach a stable state after a finite number of actions. In this stable state a_1, a_2 and a_3 have been executed[4]. Enabled input actions that expect an input from the environment, will remain enabled until that input signal is generated by the tester.

[4]Actually, each action may have been executed several times in the path from one stable state to the next.

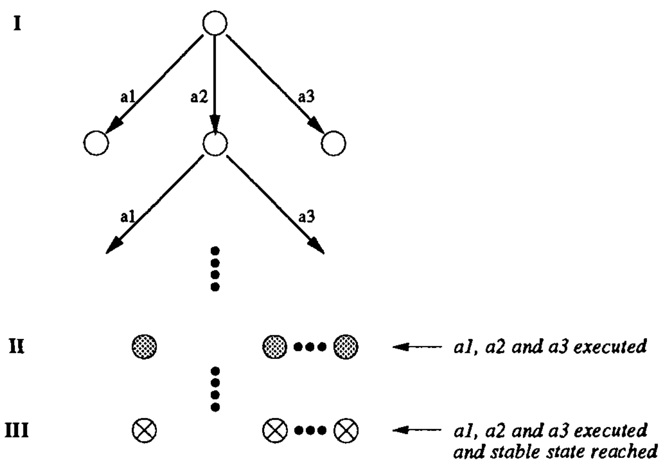


Figure 6.6: Behaviour of the global system, from one stable state to another.

The reasoning above is based in the idea that every enabled action that is not executed will remain enabled in a next state. Note that this assumption is always true for ***SDL***'88. However, if the any-construct is used that was added in the 1992 version of the language, this assumption is not necessarily true. In figure 6.7, an example is given of the use of the any-construct. Before the choice is made, both the *a*? and *b*! actions are enabled. If the a-input action is chosen, the output actions will no longer be enabled in the next composite state, even though it was not executed by the system.

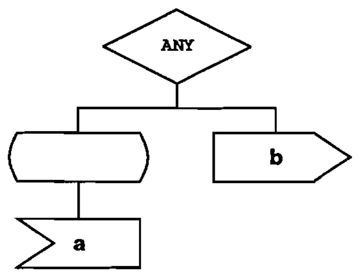


Figure 6.7: Example of the any-construct.

6.2.3 Projection sets

A path through the composite state space contains actions from different processes. A projection of a trace on a process A is denoted as $t(i) \downarrow A$. A projection of a trace $t(i)$ on a process is the trace $t(i)$, with all actions that are not executed by process A hidden (i.e. the sequence of actions that are executed by the process within the trace). All possible projections together form a projection set:

Definition 10 (Projection Set) *The set $\mathcal{P}[t(i)]$, the projection set, is defined as the set that contains all projections of the trace $t(i)$ plus a partial ordering for all elements.*

$$\mathcal{P}[t(i)] = \{t(i) \downarrow P_1, t(i) \downarrow P_2, \ldots, t(i) \downarrow P_l, \Omega\}$$

A partial ordering Ω is defined for the actions in the set. The ordering Ω is defined by the order in which the actions can be executed by each process. So there is only an ordering for actions belonging to the same process. Later in this section will be shown that this ordering can be refined to a more complete ordering to increase the fault coverage of the generated tests.

6.2.4 Controllable behaviour

Large parts of the system's behaviour can not be controlled from the outside of the system. The internal communications are "hidden" from the tester so a large part of the non-stable states in the labeled transition system will not be visible. A tester is only able to observe outputs and to send input-signals. To test the behaviour of the implementation under test, the following strategy will be used (based on the reasonable environment from section 4.6.3):

- wait until the system has reached its initial *stable* state;
- send an input signal to the system;
- wait until the system has reached a new stable state, while recording all outputs from the system;
- repeat the sending of inputs in such a way that all parts that have to be tested are executed (if possible).

Consider series of actions between two stable states: in each stable trace, at least one input action from the environment is executed (the first action). A stable trace $t(i)$ is defined for a stable state pair (s_s, s_e) as a sequence of actions that starts in s_s with an input action 'i' and ends in s_e. The following notation is used:

$$s_s \xrightarrow{t(i)} s_e$$

In the remainder of this section, only traces between stable state pairs (s_s, s_e) will be considered in which exactly one external input is allowed (necessarily the first action of the trace). In other words: only pairs of stable states will be considered such that the end-state can be reached from the starting state with only one input from the environment.

Because of the nondeterminism in the global system, the state after a sequence of actions may be one of a set of states. Therefore, the end state s_e, after input i in state s_s is not always determined deterministically. Now, the set $\mathcal{S}_e(i)$ is defined as the set of stable states in which the system may reside after external input i.

$$s_s \xrightarrow{t(i)} s_e : s_e \in \mathcal{S}_e(i)$$

For each member of $\mathcal{S}_e(i)$, there is a stable trace. However, there may be more than one stable trace for a pair (s_s, s_e). Considering a stable state s_s, the following situations can be distinguished:

1. There is only one stable state $s_e(i)$ that can be reached from the starting state after the execution of one input i from the environment. Different traces $t(i)$ may exist but the projection set $\mathcal{P}[t(i)]$ of is the same for each trace $t(i)$. For each process A:
$$t_1(i) \downarrow A = t_2(i) \downarrow A = \ldots = t_n(i) \downarrow A$$
so:
$$\mathcal{P}[t(i)] \rightarrow \mathcal{P}(i)$$

2. There is more than one stable state $s_{e_j}(i)$ that can be reached from s_s by the same external input. For each end-state s_{e_j} there exists only one set $\mathcal{P}_j(i)$.

3. There is a pair (s_s, s_{e_j}) for which there is more than one set $\mathcal{P}(i)$.

4. There are two pairs (s_s, s_{e_a}) and (s_s, s_{e_b}) for which there are traces $t_a(i)$ and $t_b(i)$ such that $\mathcal{P}[t_a(i)] = \mathcal{P}[t_b(i)]$. In this situation, all processes perform the same actions but the system may reside in to different states after the execution of the actions.

The situations that are described in items 2,3 and 4 can also be combined. A situation in which there is more than one possible end-state occurs when a single process chooses between different behaviours. There are three ways for a process to branch:

- different enabled inputs. When the process expects different inputs, each input might lead to a different behaviour. Since no external inputs are executed in the stable trace besides the first action, the branching takes place on the basis of two (or more) enabled internal inputs.

- choices on the basis of parameters that can not be accessed in the model or nondeterminism from the use of special constructs in ***SDL***'92. When a process makes a nondeterministic choice, the behaviour, and therefore the end-state, can be different.

- the expiring of a timer. When a timer expires in ***SDL***, a message is sent to the process. In a situation that the timer signal may arrive simultaneously with an input action (from an other process), there is no way to determine which signal will arrive first. From the testers point of view, the choice between both resulting behaviours is nondeterministic.

6.3 Generating tests for SDL-specified systems

Using the stable traces from the former sections, an outline will be given in this section for an algorithm that generates tests from an ***SDL***-specification. A test sequence will consist of four parts:

- a starting state;
- an input;
- a set of possible outputs;
- a set of possible end-states.

A problem, when deriving tests for ***SDL***-specified systems, is the nondeterministic delay that each output message suffers before arriving at the boundary of the system[5]. In the following section, output sets will be introduced but it will be assumed that there is no such delay. In the following section the idea will be extended when the nondeterministic delay is added to the model.

6.3.1 Output sets

The starting state and the end-state of the test sequence will be the stable states of the $(s_s, \mathcal{S}_e(i))$ pair (i.e. the end-state can be one of a set of stable states because of nondeterminism). From the projection sets $\mathcal{P}[t(i)]$ an output set is deduced. An output set is denoted as $\mathcal{O}[t(i)]$ and contains the same ordering as the projection set, but hides all actions that do not send an output to the environment (i.e. that are not observable).

Consider an arbitrary stable state s_s of the system and an external input i which is enabled in this state. In the most simple case there is a *single* output set and a *single* end-state for this input. In this situation, a simple test case can be defined for this input. The difference with the usual tests is that there is no exact sequence of actions given, only a partial ordering of actions. A way to express this partial ordering in TTCN is to write down all possible paths that exist in the specification that lead from s_s to s_e as in the representation of the state space, but a different notation could be considered.

[5]The nondeterministic delay for internal messages will be discussed later.

As noted in the previous section, it might also occur that there are two or more different output sets $\{\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_k\}$ for all possible traces $t(i)$ between a pair (s_s, s_e). In this case, the tester should expect all outputs of the different output sets. When the end-state is reached, the tester is able to determine if one of the possible output-sets was generated by the system. In this case, the stable end-state that is reached is still known. The tester will return a "`fail`" if the received outputs do not match any of the output sets. Due to the nondeterminism the tester can not just return an "**pass**"-verdict if the outputs do match one of the output sets. The specific input should be repeated several times in the same state to test the other behaviours as well. For a specific stable state s_s of the ***IUT***, the test is only then complete, when every output set $\mathcal{O}_j$ has been observed at least once.

In some cases, the choice between the different output sets can not be influenced at all (e.g. when the choice depends on a critical race). The only way to execute all different output sets, is to repeat the test (the input i in state s_s) until every output set is executed. However, when the choice depends on parameters or variables that are not visible in the model, it is the responsibility of the tester to choose the appropriate values of inputs such that every possible output set is executed. In the case that the existence of multiple output sets is a result of timers that may expire, the tester must chose the timing for the test in such a way that each output set is executed.

The same combination of s_s and i could lead to a nondeterministic choice of the system. The end-state will now be one of a finite set of stable states. The first assumption will be that the output sets for each pair (s_s, s_{e_j}) are unique (i.e. for the state s_s, each possible output set $\mathcal{O}_j$ belongs to one pair (s_s, s_{e_j})). If this is true, the outputs that are received by the tester allow the tester to determine which end-state is reached by the system. As in the previous section the tester passes the verdict "`fail`" if the response from the system does not match any existing output set for that state and input.

The situation will become more difficult when an output set, say $\mathcal{O}_z$, could indicate traces between more than one (s_s, s_{e_j}) pair. What happens is that the tester is not able to determine the end-state that is reached. It can be checked whether the sequence of outputs conforms to an output set but the continuation of the test sequence is difficult. Instead of a known state, there is a set of states, called the **uncertainty**, which holds all possible states in the the system may reside at that moment.

After the execution of a stable trace, three situations can be distinguished, depending on the uniqueness of the enabled output sets and the degree of nondeterminism. Examples of these three situations can be found in appendix E.

- There is only one possible end-state s_e.
- The end-state is one of a set $\mathcal{S}_e(i)$ but the tester is able to determine which end-state s_{e_j} is reached from the received outputs.
- The end-state is one of a set $\mathcal{S}_e(i)$ and, using the received outputs, the tester can

not determine exactly which end-state s_{e_j} is reached.

To perform a complete test on a system, at least every possible output set should be tested. This way, it is ensured that all enabled behaviour of each separate process is tested. However, this does not mean that all composite behaviour is tested. The fault coverage of the generated tests can be improved by adding an extra step. This last step checks whether the end-state s_e is actually reached. This can be done by forcing some behaviour of the system which identifies the state. However, because of the limited controllability of the system, the computation of such behaviour will be very difficult. The methods that are given in this section can not be used to generate this last step.

Because of the nondeterminism, the tester can not force arbitrary sequences of stable states. Also, it may be impossible to determine in which state the system resides after the execution of a trace. Therefore, designing complete test suites that cover the largest possible part of the system is very difficult. In [DGK91, Klo92], algorithms are presented that are able to generate complete test suites for nondeterministic finite state machines. The algorithms that are presented can be rewritten such that they can be applied to this particular problem without too much problems.

6.3.2 Observation of the output sets

Up till now, it was assumed that output sets can be observed directly by the tester. In this section the presented methods will be adapted such that the output sets represent the outputs as they arrive at the boundary of the system[6]. It is assumed that it is possible to determine a finite amount of time, after which all messages that have been sent have arrived at their destination. This is not possible within the semantics of ***SDL***, but should be possible in practical systems. This assumption is necessary because the tester must me able to determine whether a certain message has not been sent.

A problem occurs when output messages are sent to the environment via different channels. Due to the nondeterministic delay suffered by the messages on the channels, the sequence in which they are received is not necessarily the sequence in which they were sent. The interleavings of two arrivals of signals that were sent via different channels do not denote different behaviour. Therefore, there should not exist any temporal ordering between two outputs in an output set when both messages *can* be on a different channel at the same time.

An ordering should exist when messages are sent via the same channel[7]. Ordering relations between outputs can only exist per ***PCO***. Even for signals that arrive at the same ***PCO***, the ordering may be nondeterministic. Nondeterministic ordering of signals arriving at the same ***PCO*** may be caused by two constructs:

[6]Messages can be received by the tester at **points of control and observation** (***PCO's***)
[7]Outputs to the environment can only be delivered at a ***PCO*** via a channel.

- **Two channels arriving at the same *PCO*.** This kind of nondeterminism is not visible in the ***LTS***.
- **The output actions are executed by parallel processes.** These interleavings are visible in the ***LTS*** and denote different behaviour of the system.

After the output sets for a specific stable state pair have been defined, it is necessary to delete those ordering relations from the set that define an ordering for signals that arrive via different channels (either at one or multiple ***PCO***'s).

6.3.3 Refinement of the output sets

When the output sets are used as described in the former sections, the fault coverage is limited. Since the actions in the output set are only ordered with respect to actions that belong to the same process, the output sets can be used to test whether each separate process performs a valid computation. Some possible errors in the system can not be detected since the behaviour that is allowed by the output sets is not always correct behaviour according to the specification. This is due to the fact that interaction between processes restricts the possible behaviour of the system. The ordering in the output sets should be stricter to exclude all incorrect behaviour. This can be done by adding extra ordering relations to the projection sets and by splitting projection sets into multiple sets.

The effectiveness of the test generation-methods from the previous section can be improved with some small changes in the definition of the output sets. Since the output sets are derived from the projection sets, the projection sets will be redefined for a pair of stable states. A projection set for a stable state pair (s_s, s_e) was defined as the set of all possible actions that can be executed in traces between s_s and s_e, and a partial ordering Ω. The ordering existed only between actions of the same process. Suppose x_1, x_2 and x_3 are actions of the same process, than:

$$x_1 > x_2 > x_3$$

denotes that the action x_2 can only be executed *after* x_1 and *before* x_3.

An ordering, however, also exists between actions of different processes. For instance, an input of an internal message can only be executed if the appropriate output action has been executed before. When an ordering is defined that also holds relations between actions of different processes, the possible behaviour that is allowed by the projection sets is more restricted. An extended ordering for the actions in the projection set will lead to a stricter ordering in the output sets. When the implementation under test is tested against those stricter sets, the chance of finding possible errors in the implementation increases.

Each internal communication pair (i.e. the output and the accessory input action) leads to an extra ordering relations in the projection set. These input/output relations can be found in the ***SDL***-specification. For each output action, the destination process is fixed. However, when actions occur more than once, a problem arises. In this case, it is not always obvious which input actions belong to which output actions. For instance, a process may be able to receive a certain message in different states of the process. In the composite system, one or more of these inputs may actually be enabled. The ordering that is defined for the projection set should only hold relations for input/output pairs that can actually be enabled in the composite system such that:

$$output(m_x) > input_1(m_x)$$

$$output(m_x) > input_2(m_x)$$

In the labeled transition model of the composite state space, these relations can easily be found. Using these relations, the ordering Ω can be refined. In general, the ordering Ω will still be partial.

Using the composite state graph, the projection sets can be split into multiple, completely ordered projection sets. Each path that exists in the global state space, denotes different behaviour of the system. To maximize the fault coverage of the generated tests, all different behaviour should be tested. This means that every possible path between two stable states (i.e. each stable trace) should be tested. This can be done by defining a separate projection set for every possible stable trace. These projection sets will denote exactly one trace. The ordering that is defined over the actions in the set, therefore, will be complete instead of partial. Since different interleavings of internal actions will not be visible as outputs at the outside of the system, the translation to output sets will map different traces on the same output set. Still, the number of output sets that will result from this approach will grow dramatically. The ordering that is defined is to strict, since there is also an ordering defined for actions that are not ordered in the system (e.g. independent actions of concurrent processes).

6.3.4 Notation

The concept of output sets can not be expressed in ***TTCN*** directly. One solution to this problem would be to create a tree for every possible interleaving of non-ordered elements in an output set. This can easily be done in ***TTCN***, provided the trees are rewritten as described in section 4.6.1. An example of the rewriting of a tree is presented in figure 4.11. Such a solutions may, however, have some disadvantages. For instance, if there is only little ordering in the output set, the amount of possible interleavings, and therefore the width of the tree, will become very large. This results in very large test specifications and loss of overview. This overview is important if the tester wants to determine if a certain output set has already occurred or not.

A different approach seems to be more suitable for the test method based on output sets. If it is possible to define an input set in ***TTCN***, which is the complement of the relevant output set, only the input sets that can be expected have to be specified. This can not be represented in a tree because only *after* the reception of all messages, the tester is able to determine which of the possible input sets (if any) has been received. An input set could also be split into separate sets, that each represent a ***PCO***. An example of plain input sets in a test specification is given in figure 6.3.4.

```
                          Test specification
<10> PCO!msg
   <20> START T
      <30> ?[set #1]                                  PASS
      <30> ?[set #2]                                  PASS
      <30> ?TIMEOUT T                                 FAIL
      <30> ?OTHERWISE                                 FAIL
```

Figure 6.8: Test specification using input sets.

6.4 Application of the reduction methods

The reduced search algorithm that was presented by Holzmann et al. [HGP92] is able to compute paths through the global state graph such that the complete behaviour of the specified system is covered by these traces with respect to some correctness properties. The algorithm was designed for protocol verification, for which Holzmann defines the following properties that are to be verified:

- Each separate process performs a valid computation
- All processes together proceed from a known composite initial state to a known composite final state

The algorithm distinguishes *local* and *global* actions in the state graph. Local actions are actions that only refer to objects that are local to the executing process. In each state in the graph, the algorithm searches for processes which actions are all local. Such a process may execute any of the actions, without any effect on any other process. The algorithm, therefore, does only include these actions in the search path, and blocks the actions of all other processes in that state. The graph that only contains the non-blocked actions (i.e. the actions that have not been executed by the algorithm) represents the reduced state space.

In the reduced graph, there will only be one path for the different interleavings of parallel actions if these actions are independent. When parallel actions are not independent, all interleavings will be represented in the graph. For instance, actions that send a message to a queue that is able to receive other messages as well, will not be *local*.

The reduced-search algorithm assumes that each enabled action in a state can be chosen for continuation and that enabled actions can be blocked. While some transitions can be blocked in the composite state graph, they can not always be blocked in the physical system due to the limited controllability of the system. Holzmann's reduced search algorithm, therefore, can not be applied directly to automatic test generation. There are two ways in which Holzmann's reduced search method (or any state space reduction method) can be used in test generation from ***SDL***-specifications.

- **Reduction of the global state space.** If a part in the global state space is blocked such that for a pair of stable states (s_s, s_e), all paths from s_s to s_e, starting with the same input are blocked, the output set for that pair and input i does not have to be tested. Every stable trace between the pair that is specified in the projection set that starts with i, represents behaviour that is already accounted for elsewhere in the state space. If projection sets are excluded in the reduced state graph, non of the inputs in that state have to be tested.

  When the system arrives at a state that does not have to be tested at all (i.e. all projection sets for the traces that start in s_s are blocked), the easiest way is to proceed via an arbitrary input. However, this is not the most efficient way: when a state is blocked by the reduction algorithm, this is possible because the same behaviour occurs several times in the ***LTS*** and is not blocked elsewhere in the model. It is sufficient to test the behaviour elsewhere. However, it can also be tested in this state. A major improvement could be achieved when the algorithm is able to determine when blocked behaviour was not tested yet, so the test can proceed with these actions.

- **Definition of the partial ordering for actions in a stable trace.** The reduced-search method can be used to find out which interleavings of actions between two stable states denote independent behaviour and which do not. Interleavings that still exist in the reduced state space must be tested separately to achieve maximal fault-coverage. While in the former section, each path had to be tested separately, the tests can now be restricted to only the selected paths. However, since the choice between different paths can not be guided, it is not enough to mark those paths. A practical approach could be to define different projection sets for interleavings of actions that still exist in the reduced state space. Interleavings of independent actions are included in the output sets a partial orderings, so each representative of the interleaving may be selected by the system.

Both different ways of application of the partial-order reduction methods lead to different kinds of reduction. Only when complete states in the composite state space are blocked, a reduction in the memory requirements is achieved. All other states must be computed by the algorithm and must be stored in memory. If projection sets are blocked by the algorithm, this means that the testing of similar behaviour, elsewhere in the state space is sufficient for a complete conformance test. This way the number of required tests can be reduced. When the partial-order methods are used to determine the ordering that must exist between the received output messages, no reduction in time or memory requirements are achieved. However, the quality of the tests (i.e. the fault coverage) is improved.

6.4.1 Adoption of the algorithms

The reduced search algorithm that is described in section 5.3.1, was developed for protocol verification. In this section, the reduced search algorithm is used as an example for the application of the partial-order reduction methods to automatic test generation. To be able to apply the algorithm to the test generation method that was described in this section, it must be determined how the notion of *local* and *global* actions can be translated. Further it must be determined whether the correctness properties on page 81 are sufficient for a conformance testing. In other words, it must be determined that the reduction does not block actions that should be tested in a conformance test.

In the paper [HGP92] Holzmann et al. prove that, after the application of the algorithm, it can still be established that each action of every process is executed at least once. Therefore, each action will occur at least once in the reduced state graph. Secondly, a path exists in the reduced state space for each valid interleaving of dependent actions. The ordering between actions is preserved for all dependent actions. The dependence is defined with the help of a distinction between global and local actions. An action is local if it refers to objects that are local to the executing process. Objects in an *SDL*-specifications to which a process may refer are queues and variables. So, if an actions reads or writes to a queue that can only be accessed by that process, it is called local.

The reason why this distinction is made, is that interleavings of two dependent actions denote different behaviour of the system. Therefore both of the interleavings could lead to an error in the specification. The same reasoning also holds for conformance testing. So, when generating tests, both interleavings must be tested. In order to ensure that both interleavings are tested, both paths should exist in the reduced state space. Since this is the task of the reduced search algorithm, the existence of both interleavings in guaranteed. The differentiation between local and global actions is for conformance testing the same as for protocol verification.

The other algorithms that are presented (the sleep set algorithm and the stubborn set algorithm) differ in the way the dependencies are defined between the actions in the system. The reduced state spaces that follow from these algorithms still meet the demands that were stated in this section. All three algorithms are able to block different parts of a composite state space which makes them more or less suitable for specific kinds of behaviour. The stubborn set algorithm and the sleep set algorithm can be applied in the same way as the reduced search algorithm.

6.4.2 Test procedure

To use the methods from the former sections, an algorithm must be found such that the whole (or at least the largest part) of the non-blocked state graph is visited. One way could be to set up separate test suites, the same way the algorithms are applied in the PTT Conformance Kit. In this case, a test is defined for every pair of stable states and enabled input. Such a test may have to be executed several times to test each possible output set. Algorithms must.be found that force the system from an arbitrary state to a known state (**Synchronizing Sequences**) and algorithms that transfer the system from a known state to a certain new state (**Transferring Sequences**).

The computation of those sequences will be very difficult because of the limited controllability and the non-determinism. A second approach could be to make a semi-random walk through the state space. This methods resembles the "Transition Tour" test generation method. To reduce the time that is necessary to obtain reliable results, an algorithm like the one given below could be used.

1. Compute reduced composite state graph;
2. Give an input that is most likely to result in the execution of a non-blocked outputs set;
3. Find output set for generated outputs;
4. Block tested output set;
5. Repeat until the complete graph is blocked.

Both methods can be extended by using *UIO*-sequences (**Unique Input/Output sequences**) so the tester can establish whether the end-state of the stable state pair is actually reached.

The major disadvantage of both methods is that they both need a model of the complete state space. The state space explosion problem will limit the applicability of the algorithms to only small protocols.

6.4.3 Fault coverage

All tests that are generated following the proposed methods, assume sequences from one stable state to a new stable state. This is because a reasonable environment was assumed as described in section 4.6.3. This way, it can be ensured that every action of each process is executed al least once. However, an important part of the behaviour is not tested. For instance, situations in which the system might receive both internal and external messages are not tested. In the state space this behaviour is characterized by paths that execute external inputs in non-stable states. Without detailed information about the systems internal structure and timing, it is not possible to execute these traces. Therefore, this adds up to the limited controllability of the system. The fault coverage is limited by the controllability of the system. It should be investigated to which extend paths that still exist in the reduced state space can not be tested because of this limited controllability.

In cases where important behaviour is not tested because of the assumption of such a reasonable behaviour, the tester could decide to run a set of tests that send input messages to the ***IUT*** before a stable trace was finished. Most of the time it is impossible to force those traces in a deterministic way. These tests will have to be repeated several times to reach certain state transitions.

Within the limited controllability and observability, the methods that are presented here can achieve a relative high degree of fault coverage. The tester can control the coverage by the strictness of the ordering in the output sets. The algorithms that were developed in the ARTEFACT project can be used to adapt the partitioned test generation method (including the unique input/output sequences) for the test generation methods presented in this chapter.

7 Conclusions

This report started with the question if the partial-order based reduction methods that are presented by Holzmann et al. are applicable to the automatic tests generation techniques, as implemented in the PTT Conformance Kit. Also, the demand was added that the staring point would be the formal language ***SDL*** (in stead of finite state machines, which are used in the PTT Conformance Kit). The use of ***SDL*** introduces some severe problems. Before the question could be answered, a resumé of the test generation techniques that are available for ***SDL*** had to be made.

One of the typical results of the use of high-level formal description languages is the need for a translation step before the actual test generation can take place. This translation step converts the description of the protocol into an "internal representation" that contains the states in which the system may reside and information about the possible state transitions. For the creation of such an internal representation a single global system must be composed out of all separate processes. During the composition of the global system, the number of possible states explodes. Several techniques are presented to reduce the size of the generated state space such that the protocol can be analyzed automatically, within reasonable amounts of computer-resources (i.e. CPU-time and memory).

The possibilities of a conformance test are limited by the controllability and observability problem. Much of the behaviour of the implementation under test can not be controlled or observed since the access the tester has to the system is restricted to a small number of "points of control and observation". For many systems it is very difficult to achieve a reasonable fault coverage (the amount of possible errors that is found in a conformance test). Due to several constructs in ***SDL*** and the asynchronous nature of the communication of an *SDL*-system, the behaviour often is nondeterministic from the testers point of view. Nondeterminism further limits the possibilities of a conformance test.

Different approaches exist to automatic test generation from *SDL*-specifications. Most methods are based on the same sort of "reasonable environment". The assumption of a reasonable environment addresses the problem that very little information is available about the ***IUT*** when it is executing internal actions. Different kinds of environments can be defined, which differ in the restrictions they place on the behaviour of the system. In this report an environment is assumed that only sends a message to the system if all enabled internal actions have been executed. In this case, the system resides in a "stable state", which is characterized by the fact that all queues in the system are empty. Such an assumption is necessary because it is generally not possible to obtain information about internal steps that are taken by the system. For the generated tests, this means that after the application of the input message, the tester will wait until it is reasonable to assume that no outputs will be sent by the ***IUT***, without a new input signal. Such an assumption, however, does limit the fault coverage of the generated tests.

Besides the assumption of a reasonable environment, other methods are presented that can be used to reduce the composite state space (and therefore the effects of the state space explosion). Most of the methods that can be found in literature are based on heuristics, so the reduction may also involve parts of the system that should not be reduced since they are liable to contain errors. Partial-order based reduction methods, that have been presented in the field of protocol verification, do not use heuristics but equivalence between traces in the composite state space. The three reduction methods that are presented in this report are the **reduced search**, the **sleep set method** and the **stubborn set method**. The methods differ in the reduction they can achieve and the extra complexity they introduce as overhead during the test generation.

In order to apply the partial-order reduction methods to the automatic generation of conformance tests, non of the existing test generation techniques could be used. Therefore, the outline for a new test generation method was presented. This method is based on a similar "reasonable environment" as mentioned and uses partial-ordered output sets. The reduction methods from protocol verification can be used to improve three aspects of automatic test generation:

- reduction of the memory and CPU-time requirements during generation;
- reduction of the number of tests that are generated;
- improvement of the fault coverage of the tests.

The nondeterministic behaviour in the system limits the efficiency of the test procedure and the fault coverage of the generated tests. Before the generated tests are applied to the implementation under test, the data-fields of the messages that are sent must be completed. Also, additional timer information must be supplied. If enough information is available about the implementation, a major part of the nondeterminism can be deleted by the tester. Due to the asynchronous nature of the communication and some new constructs in ***SDL***'92, there will always remain some degree of nondeterminism.

The answer to the question whether the partial order reduction methods can be applied to the test generation methods as implemented in the PTT Conformance Kit should be: "No, not directly". On the one hand, this is due to the fact that the reduction methods were developed for protocol verification, thus assuming full observability and controllability. On the other hand, the use of ***SDL***, introduces extra problems that limit the possibilities of application of the methods in the Conformance Kit. However, if the test method is used that is based on output sets, partial-order reduction methods *can* be used. The reduction that is achieved with those methods for protocol verification, however, can not be achieved for conformance testing due to the limited controllability and nondeterminism in the composite system.

future research

More research is necessary to determine the effects of the assumption of a reasonable environment. In this report some effects of this assumption on the quality of the tests are summarized. However, it is not clear to which amount the assumption of a reasonable environment deteriorates the fault coverage of the tests. There is also very little knowledge of how tests can be generated that do not need the assumption of such an environment.

In this report, only an outline for a new test generation method was presented. More research is necessary to develop algorithms that implement the method. The new test generation method based on output sets resembles the ***ACT***-based method that was presented in the ARTEFACT-project. It should be investigated if the adapted algorithms for the generation of a partitioned tour based on unique input/output sequences can be used to increase the fault coverage of the new method.

An outline for the representation of the tests is presented in this chapter. Test that are generated for ***SDL***-specified systems will always contain partial orderings. Yet, partial orderings can not be represented in ***TTCN***. The efficiency of the tests and the notation could be improved if a partial ordering of actions could be represented in the test notation language.

Finally, the assumption that, in a stable state, all timers have been expired may be to strict. In the example of a call setup procedure, the call should be set-up *before* the timer expires. This usually involves the execution of external input actions in a state that is not stable in the strict sense. A direction for a solution could be to introduce stable states in which not all timers haven been expired. The full adaption of timers in the new test generation method could be the subject of further research.

Bibliography

[ADLU88] A.V. Aho, A.T. Dahbura, D. Lee, and M. Umit Uyar. An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours. In S. Aggarwal and K. Sabnani, editors, *Proceedings 8th International Symposium on Protocol Specification, Testing, and Verification (PSTV VIII)*, pages 75–86, Amsterdam, June 1988. North-Holland.

[Agh86] G.A. Agha. *ACTORS: A model of concurrent computation in distributed systems.* The MIT Press, London, England, 1986.

[BEVT89] H.V. Bertine, W.B. Elsner, P.K. Verma, and K.T. Tewani. Overview of protocol testing programs, methodologies and standards. *AT&T Technical Journal*, Jan./Feb. 1990:7–16, July 1989.

[BKKW91] S.P. van de Burgt, J. Kroon, E. Kwast, and H.J. Wilts. Automated protocol test suite production with the conformance kit. Technical Report TI-PU-91-982, PTT Research, Leidschendam, 1991.

[BKP92] S.P. van de Burgt, J. Kroon, and A.M. Peeters. Testability of Formal Specifications. In R.J. Linn, editor, *Proceedings 12th International Symposium on Protocol Specification, Testing, and Verification (PSTV XII)*, volume C-8 of *IFIP Transactions*, pages 63–77. North-Holland, 1992. also available as technical report TI-PU-92-544.

[Bri88] E. Brinksma. A theory for the derivation of tests. In J. de Meer, L. Mackert, and W. Effelsberg, editors, *Proceedings 8th International Symposium on Protocol Specification, Testing, and Verification (PSTV VIII)*, pages 343–363. IFIP WG6.1, North-Holland, June 1988.

[Cho78] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE trans SE*, SE-4(3):178–187, 1978.

[DGK91] H. v. Dam, H. Geels-Niestern, and H. Kloosterman. Test derivation from SDL specifications. Technical Report TI-IR-91-1330, PTT Research, Leidschendam, 1991.

[DHN90] E.M. Dijkerman, R.J. Helwerda, and J.C. Niestern. Computer-aided test suite generation for the remote test method. In *Proceedings of the 3th International Workshop on Protocol Test Systems*, October 1990.

[DKK91] H. van Dam, H. Kloosterman, and E. Kwast. Test derivation for standardised test methods. In *Proceedings of the 4th International Workshop on Protocol Test Systems*, pages III-3 - III-17, Leidschendam, October 1991.

[DSU90] A.T. Dahbura, K.K. Sabnani, and M. Umit Uyar. Algorithmic generation of protocol conformance tests. *AT&T Technical Journal*, Jan./Feb. 1990:101–118, July 1990.

[EK92] Jan Ellsberger and Finn Kristoffersen. Testability in the context of SDL. In *Proceedings 12th International Symposium on Protocol Specification, Testing, and Verification (PSTV XII)*. IFIP, June 1992.

[ETS92] ETSI-STC-ATM1-1004. Specification styles for SDL in order to specify testable systems, 1992. working draft.

[FB91] S. Fujiwara and G. v. Bochmann. Testing non-deterministic state machines with fault coverage. In *Proceedings of the 4th International Workshop on Protocol Test Systems*, pages III–259 - III–275, Leidschendam, October 1991.

[fsm69] *Itroduction to the theory of finite-state machines.* McGraw-Hill, 1969.

[God90] P.G. Godefroid. Using partial orders to improve automatic verification methods. In R. Kurshan and E. Clarke, editors, *Proceeding 2nd Interenational Workshop on Computer Aided Verification*, LNCS 531, pages 176–185, New Brunswick, June 18-21 1990.

[God93] P.G. Godefroid. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Extended version of article in: Proc. 2nd workshop on Computer Aided Verification*, July 1-4 1993. To be published in 1993.

[HB89] D. Hogrefe and L. Brömstrup. Tesdl: Experience with generating test cases from sdl specifications. In *SDL '89: The language at work*, pages 267–279, North-Holland, 1989. Elsevier Science Publishers.

[HGP92] G.J. Holzmann, P Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proceedings 12th International Symposium on Protocol Specification, Testing, and Verification (PSTV XII)*, page 13, Florida, USA, June 1992.

[Hog88] D. Hogrefe. Automatic generation of test cases from sdl specifications. *SDL-newsletter*, 12:34–53, June 1988.

[Hog90] D. Hogrefe. Conformance testing based on formal methods. In *Proceedings Third International Conference on Formal Desrcription Techniques (FORTE'90)*, pages 213–245, Madrid, November 1990.

[Hog91a] D. Hogrefe. Conformance testing based on formal methods. In *Proceedings Third International Conference on Formal Desrcription Techniques*, volume III, pages 207–223. Elsevier Science Publishers, 1991.

[Hog91b] D. Hogrefe. On the development of a standard for conformance testing based on formal specifications. In J. Kroon, R.J. Heijink, and E. Brinksma, editors, *Proceedings of the 4th International Workshop on Protocol Test Systems*, pages 59–66. IFIP WG 6.1, Elsevier Science Publishers, October 1991.

[Hol87] G.J. Holzmann. On limits and possibilities of automated protocol analysis. In *Proceedings 7th International Symposium on Protocol Specification, Testing, and Verification (PSTV VII)*, volume VII, Zurich, May 1987.

[Hol89] G.J. Holzmann. Validating sdl specifications: an experiment. In *Proceedings 9th International Symposium on Protocol Specification, Testing, and Verification (PSTV IX)*, page 11, Enschede, June 1989. IFIP WG 6.1.

[Hol91a] G.J. Holzmann. *Design and validation of computer protocols*, chapter 9, Conformance testing, pages 189–205. Prentice Hall, 1991.

[Hol91b] G.J. Holzmann. *Design and validation of computer protocols*, chapter 11, Protocol validation, pages 217–246. Prentice Hall, 1991.

[Hol91c] G.J. Holzmann. *Design and validation of computer protocols*, chapter 8, Protocol validation, pages 163–188. Prentice Hall, 1991.

[Hol92] G.J. Holzmann. Practical methods for the formal validation of sdl specifications. *Computer Communications*, 15, no. 2 march 1992:129–134, march 1992.

[IC92] ITU-CCITT. Revised Recommendation Z.100, CCITT Specification and Description Language (SDL). International standard, CCITT, October 1992.

[ISO88] ISO-DIS-8807. LOTOS, A formal description technique based on the temporal ordering of observational behaviour, 1988.

[ISO89] ISO-DIS-9074. Information Processing Systems - Open Systems Interconnection - Estelle - A Formal Technique Based on an Extended State Transition Model, May 1989.

[ISO90] ISO-DIS-9646. Conformance testing Methodology and Framework, 1990.

[Klo92] Hans Kloosterman. Test derivation from non-deterministic finite state machines. In *Proceedings of the 5th International Workshop on Protocol Test Systems.* IFIP, September 1992. also technical report TI-PU-93-873.

[Koo91] C.J. Koomen. *The design of communicating systems.* Kluwer Academic Publishers, Dordrecht, 1991.

[Kri89] K. Kristoffersen. Conformance testing based on sdl specifications. *SDL '89: The language at work*, pages 257–266, 1989.

[LBDW91] G. Luo, G. v. Bochmann, A. Das, and C. Wu. Failure-equivalent transformation of transition systems to avoid internal actions. Technical Report 789, Université de Montréal, September 1991.

[LCL87] F.J. Lin, P.M. Chu, and M.T. Liu. Protocol verification using reachability analysis. the state explosion problem and relief strategies. *Computer communications review*, 17, No. 5:126–135, 1987.

[LDB91] G. Luo, A. Das, and G. v. Bochmann. Test selection based on sdl specifications with save. In *SDL '91: evolving methods*, pages 313–324, North-Holland, 1991. Elseviers Science Publishers.

[LDB93] G. Lou, A. Das, and G. v. Bochmann. Generating tests for control portion of SDL specifications. In *Proceedings of the 6th International Workshop on Protocol Test Systems*, 1993. To be published later this year.

[LL92] Lai and Leung. An industrial perspective on academic protocol testing methods. In *Proc. 11th Int. Conference on Computer Communication*, volume 1, pages 237–242, Genova, 1992.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* Lecture Notes on Computer Sciences 92. Springer Verlag, 1980.

[NH84] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[P5493] ISO SC21 WG 1 P54. Formal methods in conformance testing, june 1993. working document.

[PRO92] PROVE. CATG: Enhanced methodology for computer aided test generation. deliverable 87/EC/JHH/DS/B/063/A1, PROVE RACE Project R1087, 1992.

[SD88] K. Sabnani and A. Dahbura. A protocol test generation procedure. *Computer Networks & ISDN systems*, 15(4):185–297, 1988.

[ULS90] M. Umit Uyar, A. Lapone, and K.K. Sabnani. Algorithmic verification of isdn network layer protocol. *AT&T Technical Journal*, Jan./Feb. 1990:17–31, 1989, July 6 1990.

[Ura91] H. Ural. Formal methods for test sequence generation. *Computer Communications*, 15, no 5.:311–325, March 1991.

[Val90] A. Valmari. A stubborn attack on state explosion (abridged version). In R. Kurshan and E. Clarke, editors, *Proceeding 2nd Interenational Workshop on Computer Aided Verification*, pages 157–165, New Brunswick, June 18-21 1990.

[VT91] A. Valmari and M. Tienari. An improved failures equivalence for finite-state systems with a reduction algorithm. In *Proceedings 11th International Symposium on Protocol Specification, Testing, and Verification (PSTV XI)*, Stockholm, 1991.

Appendix A

The reduced search method

The task of the algorithm is to decide during a depth first search which edges (and succeeding paths) do not have to be explored. Typically many paths through the state space concist of the same transactions. The only differency between the paths is the sequence in which the transactions are executed. If those transactions are local to the executing process, the result of the verification will be the same for all paths. The algorithm will block paths that do not have to be explored. The following definitions will be used:

local edge: Any edge in the state graph that corresponds to a transition referring only to objects local to the executing process.

global edge: Any edge that is not local.

true edge: An edge with its eligibility flag set to TRUE.

true node: A node with its node flag set to TRUE.

search stack: Ordered set of nodes.

edge set: An edge set of state s contains edges exiting from s such that only edges that correspond to transitions from the same process are in the same edge set.

local edge set: An edge set that only contains local edges.

The state-graph representation is extended with an **eligibility flag** for each edge in the graph. An edge is called "TRUE" if its eligibility flag is TRUE. Initially, all edges are TRUE. Furthermore, **edge sets** are defined for each newly generated node. With these definitions, a **free edge set** can be defined:

Definition A.1 (Free Edge Sets) *An edge set $\mathcal{E}$, belonging to state s, is "free" iff:*

- *$\mathcal{E}$ consists of only* local *edges;*
- *$\mathcal{E}$ contains at least one TRUE edge, "e";*
- *the successor of s via e, was not already visited.*

```
reduced_search_algorithm

$\mathcal{S}$ := {initial_state};                          /* search stack */
$\mathcal{D}$ := $\emptyset$;                              /* visited states */
ALL eligibility flags := TRUE

DO  WHILE $\mathcal{S} \neq \emptyset$
    $s$ := last_element_of $\mathcal{S}$;
    create_edge_sets for $s$;
    IF $\exists$ (TRUE edge from $s$ to successor $s'$ outside $\mathcal{S}$ and $\mathcal{D}$)
          IF $\exists$ (free edge set for $s$)
                mark all edges in all other edge sets of $s$ FALSE;
                add $s'$ to $\mathcal{S}$;
          ELSE  select TRUE edge from $s$ to $s'$;
                add $s'$ to $\mathcal{S}$;
          FI
    ELSE  remove $s$ from $\mathcal{S}$;
          add $s$ to $\mathcal{D}$;
    FI
OD
```

Figure A.1: Partial search algorithm

When the algorithm arrives at an unvisited node n in the state graph it wil explore the nodes that can be reached via an enabled transaction from n. In certain cases the algorithm will decide not to explore all possible successors:

1. Edges that lead to successors that are not TRUE (that have already been visited) will not be traversed.

2. If an enabled local edge set $\mathcal{E}$ exists (first IF statement in the algorithm) the algorithm will only traverse the edges in $\mathcal{E}$. If more than one of such sets exist, one set is selected arbitrarily (e.g. the one leading to the node with the lowest ordeal number).

A Proviso

The search algorithm tries to find parts of behaviour that can be executed independently from the rest of the system. If this behaviour is found, the rest of the behaviour is temporarily blocked. Holzmann defines the proviso that must be met to allow the blocking of edges in a state:

"There exists a local edge exiting from the current node that leads to a node outside set $\mathcal{S}$" (the search stack).

For the proper reduction of the state space, it is also necessary that the search is continued via the edge set to which this local edge belongs. The next example will show that for a proper reduction of the state space, it is necessary that the search is continued via an edge set that contains al least one TRUE edge that leads to a node outside $\mathcal{S}$.

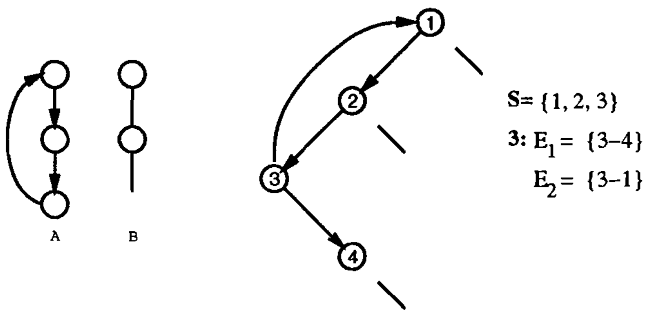


Figure A.2: Small example combining two independent processes A and B. S = search stack and E are the edge sets.

The example in figure A.2 shows that in this situation, a proviso of a kind is necessary: if the first edge set (containing edge 3-1) would be selected for the continuation of the search, all paths that would allow transactions of process B would be marked ineligible. As a result of this, not all processes perform a valid computation and the algorithm would not meet our demands.

Proof for the reduced search method

LEMMA: When a state s is removed from $\mathcal{S}$, all transactions that each of the processes in state s could execute have been executed, either at state s itself, or at successor states of s.

PROOF: Consider s_0; the first state that is backtracked during the search. It is the last state of the first path explored by the modified depth-first search. Since there are no successors for s_0, the lemma holds for this state.

Next, state s_n will be considered. This is the n-th state to be backtracked. There are two possible cases:

1. All enabled transitions of s_n have been executed;
2. All edge sets except one, set $\mathcal{T}$, are marked ineligible and all transitions from $\mathcal{T}$ are executed.

The lemma holds (trivially) for the first possibility. Now suppose the lemma holds for s_{n-1}:

XXX

define $\mathcal{T}'$:= all enabled transactions not in $\mathcal{T}$.
s' := a state that is a successor of s_n via a true edge in $\mathcal{T}$ and s' not a member of $\mathcal{S}$.

There must at least be one state s' and one true edge in $\mathcal{T}$ that leads to s'. Since the transactions in set $\mathcal{T}$ belong to an other process than the ones in $\mathcal{T}'$, all transitions from $\mathcal{T}'$ remain enabled after an execution of a transition in $\mathcal{T}$. Therefore all transitions in $\mathcal{T}'$ are enabled in sate s'.

From the assumption that the lemma holds for s_{n-1} and the fact that it is known that all successor states of s_n have been backtracked, follows that all transitions in $\mathcal{T}'$ have been executed. Since it is known that all transactions in $\mathcal{T}$ were executed in the normal depth-first search, all enabled transactions in s_n must haven been executed. By induction this proves the lemma.

Applying the lemma to the root node of the global search tree proves that with the proviso, all reachable transitions will be executed at least once in algorithm 2.

Figure A.3: Proof for the reduced search method.

Appendix B

Conflict set method

The description of a protocol holds the behaviour of the system. Often, this behaviour is expressed as a reaction to stimuli from the environment. Therefore, most actions in a protocol specification will not be local ones. Still not every global action is dependent. Reduction can be achieved by determining which global actions are dependent, and which are not. The conflict set method uses a conflict set to determine is a dependent action has occurred between now, and the last occurrence of the action. Note that the conflict set method is based on dependencies of actions of competing processes an not on dependencies between state transitions (as in the reduced search method).

Conflict sets

Holzmann et al. introduce conflicts sets that are used to determine if the execution of a statement must be repeated in the depth-first search. Actions should be investigated repeatedly only if a dependent action could be shuffled ahead of the one considered. The model of the composite search space is extended with conflict sets according to figure B.1. Note that the conflict sets do not belong to the state transitions, but to the actions that are tagged to the state transitions.

Conflict sets are assigned to every action that can be executed. For statements that occur more than once in a process definition, a conflict set is assigned to every single occurrence of the statement. If code (e.g. in macros) can be executed by multiple processes, conflict sets are assigned for every process. A conflict set can contain zero or more '*conflict tags*'. Non-empty conflict sets block the execution of the statement everywhere in the composite behavior until the conflict set is cleared. This happens if one of the tags in the set 'conflicts' with another executed statement.

Conflict Tags

Up till now it is not defined when a created conflict tag should clear a conflict set or how those conflict tags are created. The first tag that can be defined is the `local`-tag. This tag is created by the execution of a statement that refers to a local variable or a

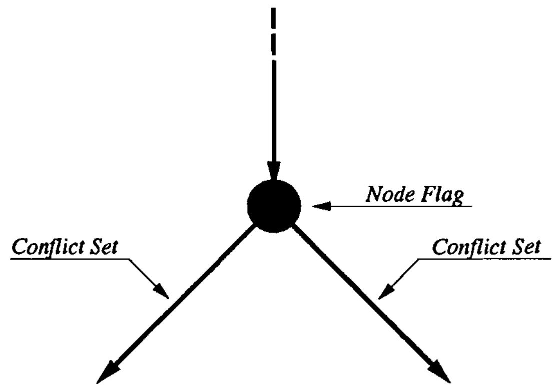


Figure B.1: A sample node in the global state tree

symbolic constant.

Reading from or writing to a global variable clearly can cause dependencies between processes. Therefore `read` and `write`-tags are created for each global variable. Receiving from or sending to message queues can also cause dependencies between processes, although this depends on the status of the queues. Conflict tags related to the sending or receiving of messages are `send` and `receive`. Consider the following example:

> Assume that x and y are global variables. The following example will create conflict tags for references to those variables. The expression:
>
> $$x := x + y + 3$$
>
> creates the tags: `write_x`, `read_x`, `read_y` and `local`. The `local`-tag is always cancelled if any other tag exists.

If a '-' denotes no dependency between two tags, a '+' denotes a dependency and an 'X' a conditional dependency, the following table shows the dependencies between all possible conflict tags. This means that, when a tag is created, all conflict sets that contain a dependent tag (according to the table) are cleared.

	local	read	write	send	receive
local	-	-	-	-	-
read	-	-	+	+	+
write	-	+	+	+	+
send	-	+	+	+	X
receive	-	+	+	X	+

Statements that belong to the same process are always assumed to be dependent.

Therefore, the execution of a statement of a process $\mathcal{A}$ clears minimally the conflict sets of all other statements that belong to the process.

Conflict Set update rules

Essentially the algorithm performs a depth-first search. The difference is that some transitions can be (temporarily) blocked during the search. When a certain global state is reached, all transactions belonging to one of the enabled processes in that state are executed first. The tags that are generated by those transactions are only inserted in the appropriate sets after the complete search through the state space after the corresponding edges is completed. So the tags are inserted into the conflict sets just before the search continues with a new process (assuming there is one).

Now, the following situation is assumed: at a point in the depth-first search as state s_i is reached. In this state $\mathcal{N}$ processes can execute one or more statements. Processes $\mathcal{P}$ and $\mathcal{Q}$ are part of those $\mathcal{N}$ statements. The updating of the conflict sets has to conform to the following rules:

- First, all statements belonging to one process in s_i will be considered (e.g. process $\mathcal{P}$);
- Conflict *Tags* are **created** upon execution of each statement, before the traversal of the corresponding edge;
- Conflict *Sets* are **cleared** immediately after the creation of the *tags*. *All* transactions that conflict with one of the enabled transactions of process $\mathcal{P}$ are now cleared;
- The depth-first search is now continued for all successors of s_i via edges from sate s_i that belong to process $\mathcal{P}$. Only when this part of the search is completed, the created conflict tags are entered in the appropriate sets. In other words: the conflict tags belonging to the enabled statements of process $\mathcal{P}$ are **inserted** just before the next process (e.g. process $\mathcal{Q}$) is considered;
- The effect of every conflict set update (clearings and entries) are cancelled (reversed) when the depth-first search 'back-tracks' to a previous state;
- When the complete behavior following execution of a statement of process $\mathcal{Q}$ is explored, the conflict tags belonging to this process in this state are inserted in the conflict sets. At this moment the conflict sets of all transactions belonging to processes $\mathcal{P}$ and $\mathcal{Q}$ are filled with the tags that are created upon execution of all possible transactions from those processes in state s_i;

examples

The update rules will now be illustrated with the help of two examples. In the example in figure B.2 two processes (A and B) are drawn. The actions of both processes that can be interleaved in any way. This results in the complete state space as shown in the figure.

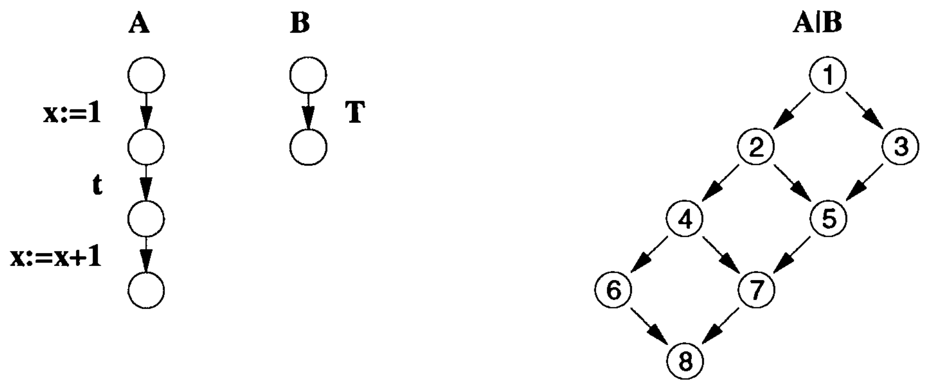


Figure B.2: Two processes A and B and their complete composite state graph.

Assume that transaction T of process B can be either dependent or not dependent of transactions touching the global variable x. It is assumed that a tag 'T_tag' is generated. Starting in node 1 of the state space, the depth-first search will arrive via the path 1-2-4-6-8 at node 8. At this point the situation is as follows:

current node	8
search stack	$\{1, 2, 4, 6, 8\}$

statement	Conflict Set	generated
'$x := 1$'	$\emptyset$	`write_x`
t	$\emptyset$	`local`
'$x := x + 1$'	$\emptyset$	`write_x`, `read_x`
T	$\emptyset$	`T_tag`

Since no more transactions can be executed, the depth-first search will back-up to the previous node. Since all possible transactions in state 6 have been executed, the search backs-up to state 4. In state 4 two processes exist (A and B). All states that are reachable from state 4 via transactions of process A are now explored. At this moment the created conflict tags for the statements (only '$x := x + 1$') must be inserted:

current node	4
search stack	$\{1, 2, 4\}$

statement	Conflict Set	generated
'$x := 1$'	$\emptyset$	write_x
t	$\emptyset$	local
'$x := x + 1$'	{ write_x, read_x }	
T	$\emptyset$	T_tag

The search will now continue with the T action of process B. Two possible cases could apply to this situation:

1. Transaction T does not create a tag that conflicts with any of the tags in the conflict set of '$x := x + 1$' (i.e. x is not accessed). The search will advance to node 7. Since the conflict set of '$x := x + 1$' is non-empty, the transaction is blocked and the search path ends in state 7.

2. By executing T, a tag is created that conflicts with one of the tags is the conflict set of '$x := x + 1$'. This causes the conflict set to be cleared. Now the search proceeds from state 4 via state 7 to state 8 (state 8 will not actually be visited since it was already marked by the depth-first search.

Continuing the search in the same way as described, the transaction t and '$x := 1$' (depending on the dependency with T), both of process A will be blocked in the rest of the search. This results in the search paths in figure B.3.

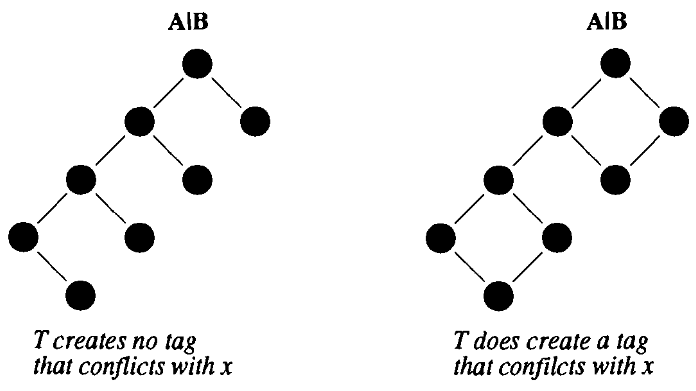

A|B
A|B
T creates no tag that conflicts with x
T does create a tag that confilcts with x

Figure B.3: complete composite state graph of processes A and B.

In most experiments that are described by Holzmann, the best results are obtained by combining both the reduced search algorithm (algorithm 2) and the conflict sets. This is particularly true for systems that contain many cycling and independent processes. The example in figure B.4 is an example of such a system.

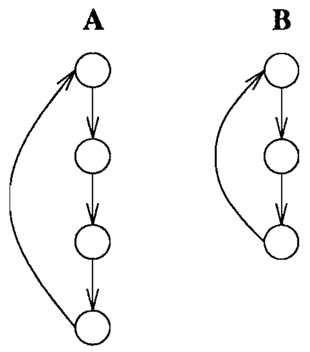


Figure B.4: example with two independent, cycling processes A and B.

Assume that both process A and B only contain local transitions. The application of algorithm 2 or 3 alone cannot prevent that during the search multiple interleavings of transactions of both processes are explored. Since the processes are independent, this would not be necessary. Both algorithms can be combined quite easily by simply applying one after the other. This can be done because the algorithms work at different points in the search: algorithm 2 can block *edges* exiting from a node when this node is reached for the first time in the search. Algorithm 3 blocks *statements* somewhere in the rest of the search. In Holzmann's paper it is assumed that applying both algorithms in the search still allows to check for the two correctness criteria given on page 16. There is (yet) no proof of this assumption. The result of applying both algorithms is shown in figure B.5. The figure shows the different search paths when algorithms 2, 3 and both algorithms were applied during the search. Dashed lines are enabled transactions that end in a state that was already visited.

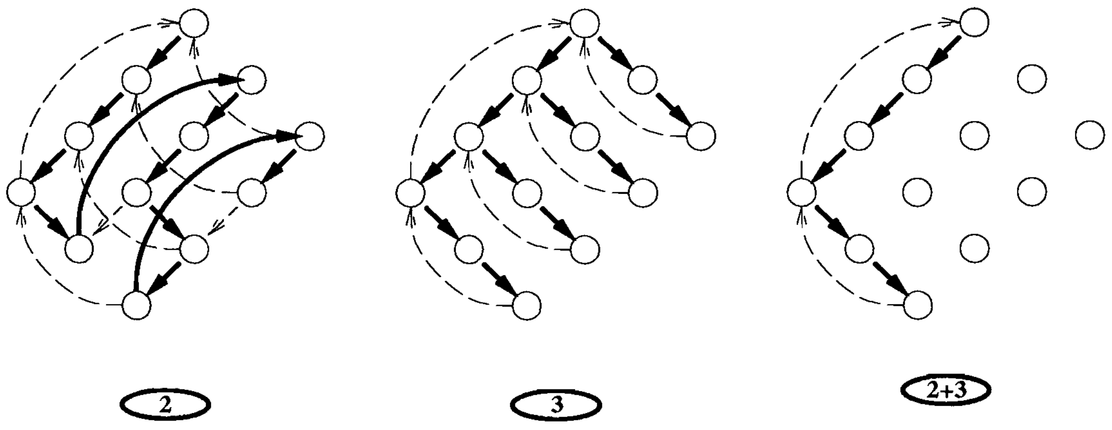


Figure B.5: Different search paths when algorithms 2, 3 and 2+3 are applied during the search.

Appendix C

Stubborn sets

A reduction of the state space can also be achieved by using the information about which processes use which variables. Methods that use these properties have been described by Overman and Valmari [Val90, VT91]. The idea of those methods is that, for some global variables, there are processes that cannot touch this variable. Statements of those processes are independent of statements that use the variable (if the statement uses no other variables). If one or more processes are about to execute a statement using this variable, the different interleaving of these transactions with the independent ones will not give additional information. Therefore, it is sufficient to explore only these transactions.

Consider tree processes A, B and C. Suppose A never touches the global variable x and that figure C.1 is a part of the composite state graph:

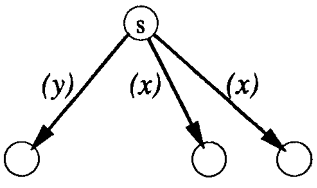


Figure C.1: a sample node in the state graph. '(x)' denotes a transaction that touches variable x.

In the figure each edge corresponds to a different process. If statement (y) is a statement of process A, only the rightmost edges would be selected to explore. In this way the interleavings of the dependent actions is generated first. The independent behavior can be executed afterwards.

Overman's method

A method to automate the reasoning that was explained in the previous paragraph was developed by Overman (1981). It uses the following definitions:

- **support set:** the set of global variables that can be accessed by a process;
- $\boldsymbol{v_i}$: a global variable;
- $\boldsymbol{V_i}$: set of variables. Contains v_i and all global variables that are touched by enabled statements that are dependent to v_i;
- $\boldsymbol{P_i}$: set of processes (for each variable) that may execute a statement. This means that processes not in the set are not dependent on the variable i. Sets V_i and P_i are created for each state that is encountered in the depth-first search.

The following algorithm computes the P_i and V_i sets for each variable at each state that is encountered during the depth-first search. The complexity of the algorithm suggests a major increase in overhead, induced by the algorithm.

1. set V_i to $\{v_i\}$;
2. set P_i to $\emptyset$;
3. for each process whose support intersects V_i, add the global variables that the *next* transitions in this process can touch to set V_i and add the process to P_i;
4. repeat previous step until no more variables can be added;
5. choose set P_i that corresponds to the smallest non-zero number of enabled transitions.

At the end of the algorithm a set P_i exists for the current state. This set contains all processes that have transitions that touch variable v_i and all processes that have transitions that touch one of the variables that can be touched by enabled transitions of processes in P_i. All processes that contain a transaction that is dependent on one of the enabled transactions of the processes in P_i are also contained in the set. Therefore it is save to restrict the search to the enabled transactions in a set P_i only.

Example

The following example specification (from [HGP92]) contains three processes. In the system there are three global variables $\{x, y, z\}$.

The reduced search algorithm would yield no optimization here. To be able to compare Overman's method to the method using conflict sets, both reduction strategies were applied to the example in figure C.2.

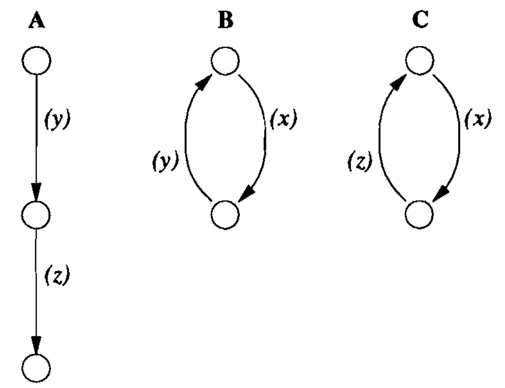


Figure C.2: Example with three processes. Note that (v) denotes a transaction that 'touches' variable v.

Stubborn Sets

An other approach to the state space explosion problem was described by Valmari. He uses the same starting points as Overman in the method that was described in the previous section. Valmari's algorithms are able to reduce the exponential state spaces to polynomial ones in some cases. Since Valmari's methods are described in more detail and for more general cases, this report will focus here on these methods.

Assume a system consisting of a finite set $\mathcal{V}$ of *variables* and a finite set $\mathcal{T}$ of *transactions*. The set $\mathcal{S}$ contains all possible states (thus the Cartesian product $\mathcal{T} \times \mathcal{V}$). Like in the previous sections:

$$s \xrightarrow{t} s'$$

is used to denote that in a state s, transaction t is enabled and, if executed, transfers the system to state s'. There is a function T: $\mathcal{S} \times \mathcal{T} \longrightarrow \mathcal{S}$; the transfer function and a initial state $s_0 \in \mathcal{S}$. The value of a variable v at state s is denoted by: $s(v)$.

To explain the stubborn set method, some concepts have to be defined. Assume $t, t' \in \mathcal{T}$, $s, s' \in \mathcal{S}$, $V \subseteq \mathcal{V}$ and $T \subseteq \mathcal{T}$.

en(s,t), t is enabled in s iff

$\exists_{s'} : s \xrightarrow{t} s'$.

en(s,t,V), t is enabled with respect to V at s iff

$\exists_{s'} : en(s', t) \wedge \forall_{v \in V} : s'(v) = s(v)$

The definition states that, if en(s,t,V) is true, there is a state s' where t is enabled and all values of the variables in V are the same as in s. Notice that if t is *not* enabled with respect to V, it is necessary to change at least one variable to enable t. From the definition follows: en(s,t) $\Rightarrow$ en(s,t,$\mathcal{V}$)[1].

T=wrup(t, V), T is a write up set of t, with respect to V iff

$\forall_{t',s'} : \neg(: en(s,t,V) \wedge s \xrightarrow{t'} s' \wedge en(s',t,V)\,) \Rightarrow t' \in T.$

A write up set of a transaction is *any* set that contains at least those transactions which may enable this transaction with respect to a set V. It is not necessary the smallest set. A write up set always exists (e.g. all transactions: $\mathcal{T}$).

t $\leftrightarrow$ t', t accords with t' iff

$\forall_s : en(s,t) \wedge en(s,t') \Rightarrow \exists_{s',s_1,s_1'} : s \xrightarrow{t} s_1 \xrightarrow{t'} s_1' \wedge s \xrightarrow{t'} s' \xrightarrow{t} s_1'.$

Two transactions accord with each other if their support sets[2] are disjoint. They also accord with each other as long as the transactions do not write to common variables. Transitions writing to FIFO queues accord with transitions reading from them. Transitions that correspond to different locations in a sequential process accord with each other because they are never simultaneously enabled.

T is semi-stubborn at s iff

$\forall_{t \in T}$:

(1) $\neg en(s,t) \Rightarrow \exists_V : \neg en(s,t,V) \wedge wrup(t,V) \subseteq T$

(2) $en(s,t) \Rightarrow \forall_{t' \notin T} : t \leftrightarrow t'$

The result of the first part of the definition of a semi-stubborn set is that disabled transitions in a semi-stubborn set can be enabled only by transitions from the same set. The second part guarantees that transactions in a semi-stubborn set accord with transactions outside this set.

Both $\mathcal{T}$ and $\emptyset$ are semi-stubborn, thus a semi-stubborn set can be found for every transition. Furthermore, if a transaction occurs outside T, the set will remain semi-stubborn.

THEOREM: if $T \subseteq \mathcal{T}$ $s_0 \in \mathcal{S}$ such that T is semi-stubborn at s_0, $n \geq 1$ and

$\sigma = s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots \xrightarrow{t_{n-1}} s_{n-1} \xrightarrow{t_n} s_n$

be a execution sequence such that:

$t_1, \ldots, t_{n-1} \notin T$ and $t_n \in T$ then:

There is a finite execution sequence σ':

$\sigma' = s_0 \xrightarrow{t_n} s_0' \xrightarrow{t_1} s_1' \cdots \xrightarrow{t_{n-1}} s_{n-1}$

such that $s_{n-1}' = s_n$.

[1]In [VALM:90] en(s,t) $\Leftrightarrow$ en(s,t,$\mathcal{V}$) is used. This is only true if no *local* transactions are possible.

[2]The support set of a transition is the set of all variables the transition can touch. Valmari uses the term 'reference sets'.

Proof of the theorem is given in earlier papers by Valmari. To use the semi-stubborn set, one property has to be added to exclude the possibility that the set does not contain any enabled transactions. This addition yields the stubborn set:

***T* is *stubborn* at *s* iff**
T is semi-stubborn at $s \wedge \exists_{t \in T} : en(s,t)$

So a stubborn set is a semi-stubborn set which contains at least one enabled transition. Since $\mathcal{T}$ is a stubborn set for every transition, a stubborn set always exists.

Construction of reduced state spaces

The stubborn sets that were presented in the previous section can very effectively be used to create a reduced state space. Let

$$\Phi(s) : \mathcal{S} \longrightarrow T$$

be the 'stubborn set generating function' such that:

$$\begin{aligned} \neg\, en(s,t) &: \ \Phi(s) = \emptyset \\ en(s,t) &: \ \Phi(s) = \text{a stubborn set.} \end{aligned}$$

Assume that a state $s \in \mathcal{S}$ has just been generated. The difference with a normal spate space generation (or exploration) is that, in stead of all possible transitions, only the enabled transitions in $\Phi(s)$ are used. Depending on the structure of the participating processes, this will yield less transactions to be executed and possibly less states to be visited.

Effectiveness of the stubborn set algorithm

The effectiveness of the presented methods is largely dependent on the stubborn set generator $\Phi(s)$. The lower boundary of the performance is the 'normal' (exponential) state space. This results can alway be achieved simply by taking $\Phi(s) = \mathcal{T}$ for all $s \in \mathcal{S}$. If $\Phi(s)$ is smaller than $\mathcal{T}$ for certain states, a reduction is possible. Valmari shows in previous papers that always taking the smallest set during the construction does not necessary yields the smallest state space.

A stubborn set is said to be 'optimal' if there is no subset of the set that is still stubborn. Algorithms are available to compute "fairly good" stubborn sets in linear time with respect to the number of transitions. Optimal stubborn sets can, under some assumptions, be computed in quadratic time (again with respect to the number of transitions).

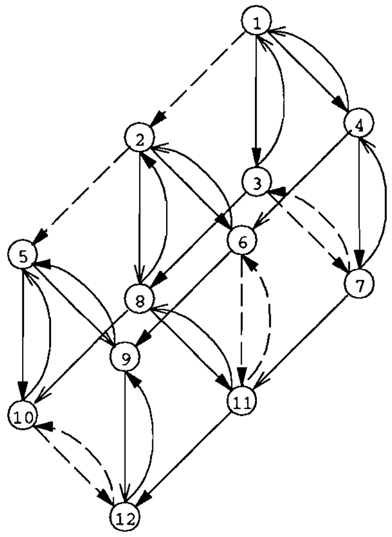


Figure C.3: Complete state space. Dashed lines denote blocked transactions after application of Overman's method.

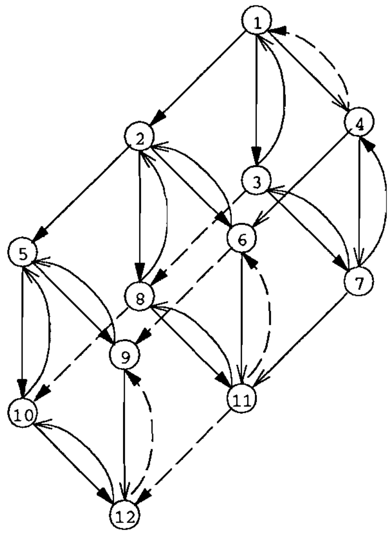


Figure C.4: Complete state space. Dashed lines denote blocked transactions after application of the conflict set method.

Appendix D

Two independent processes

This example contains two completely independent processes. The composite state space (i.e. the set of all states that can be reached by the total system) will at least consist of the Cartesian product of both individual state spaces. Depending on the communication model that is used, more different states can be reached. A conventional test generation algorithm will test every possible transition in the state space. It will be shown that this is not always necessary, and that the number of generated tests can be decreased by using this observation.

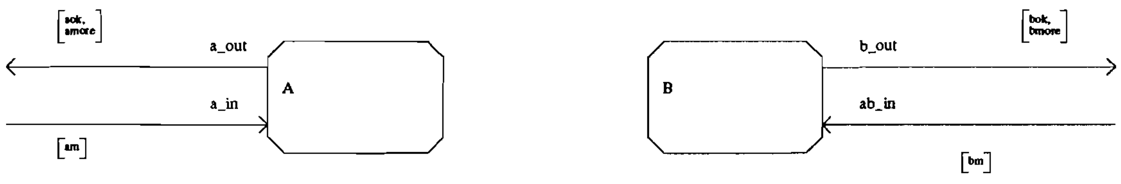


Figure D.1: A system containing only two independent processes "A" and "B".

description of the example

The first example consists of two independent processes. Both processes are deterministic and finite. The processes both perform the same "task" (figure D.2).

For the use of the PTT Conformance Kit, the example should be specified as a single finite state machine. This ***FSM***is given in figure D.3. Since no communication is exchanged between the two processes, a composite ***FSM*** is easily created (figure D.4). The **partitioned tour** that is generated by the toolset tests *every* transition in the composite state space is tested. This means that even for this extremely simple system, eight complete test cases are necessary.

- Consume a message "m";
- Prompt for "more";
- Consume next message "m";
- End with "ok";
- Start all over again.

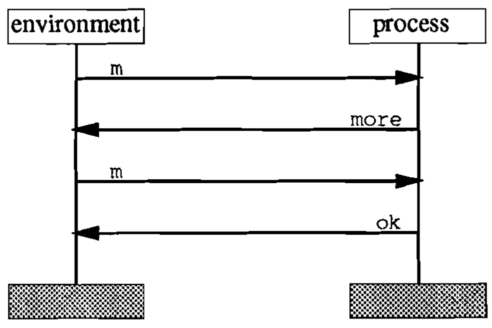


Figure D.2: Definition of a single process.

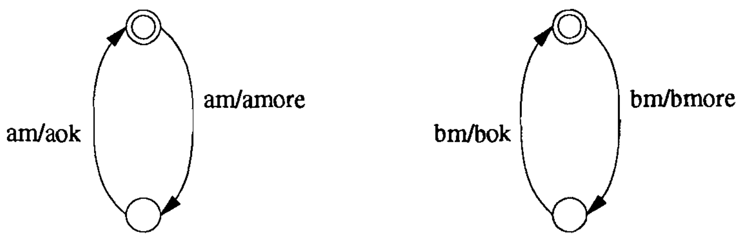


Figure D.3: The two processes independent that build the example protocol.

It can be questioned whether all these test cases are necessary to ensure the correct operation of this particular protocol. For instance, test cases that test transition (0→2) and transition (1→3) (see figure D.4) actually both test the same transition of process **B**. Whether process **A** has performed some behaviour before, has no influence on the correct operation of process **B**. Therefore, if it is assumed that the implementation has the same *structure* as the specification, the complete test suite for the system should only contain a test case for one of both transitions.

Application of the partial order methods

The partial order reduction methods that are presented in [HGP92] assume that the protocol is specified as a labeled transition model ***LTS***. Each state transition of a finite state machine actually consists of two state transitions (one for the input and one for the output) and an extra intermediary state in the ***LTS***. The methods that were defined for labeled transition systems can be applied to finite state machine-based test generation techniques in two ways:

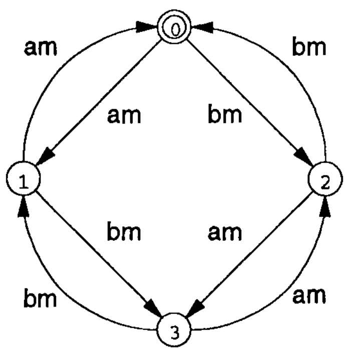


Figure D.4: The composite state space of the example.

- Convert the ***FSM***-representation of the specification to a labeled transition system. In order to generate tests from the reduced specification, the ***LTS*** should be translated back into an ***FSM***. Application of the new methods via a translation to a ***LTS*** will be called *indirect application* here. Special care must be taken if multiple transitions are allowed to start from the intermediary states since this behaviour will often be difficult to control by the tester.
- Translate the concepts of *dependency* between actions to dependency between state transitions in finite state machines. This way will be called *direct application*. When considering communicating ***FSM***'s this may not be possible.

Direct application

In this case (completely independent processes), the second possibility seems the most appropriate. Considering the ***FSM***, it is clear that any transition of process **A** is independent from any transition of process **B** and vice versa. In terms of the reduced-search algorithm [HGP92], this means that all *state transitions* are **local**.

In figure D.5, the resulting state spaces after the direct application of the reduced search method and the sleep set method to the example are shown. It can easily be verified that the blocked transitions (the dashed branches in the graphs) do not represent extra information about the system's behaviour, after the remaining transitions have been tested, and if it is assumed that the structure of the implementation is equal to the structure of the specification.

This example could suggest that direct application can by applied easiliy. However, if the processes in the specification communicate with each other, as in practical situations, the definition of atomic input/output pairs and the discrimination between global and local actions will be very difficult. This report focusses on the indirect approach, which is described in the next section.

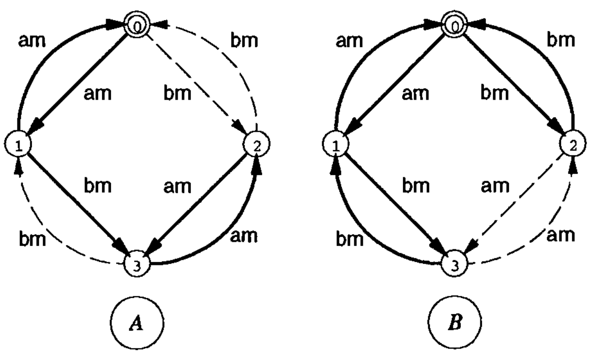


Figure D.5: Reduction of the state space. Drawing A is the result of the Reduced Search algorithm, drawing B of the Sleep Set algorithm.

Indirect Application

An other possible way to use the reduction methods for automatic test generation is via a translation to a labeled transition system. The labeled transition system is given in figure D.6. However, the behaviour that is defined by the ***LTS*** is not the same as the behaviour that was defined by figure D.4. The reason for this is that the input/output pair is no longer atomic. In the ***LTS***-model, process **B** may execute an action after process **A** has consumed an input, but before it has been able to sent the relevant output.

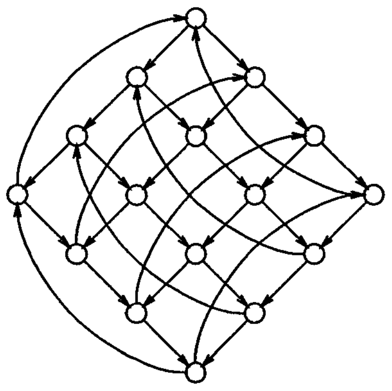


Figure D.6: State graph for the labeled transition system for two completely independent processes. Labels and reset transitions are not given.

This ***LT***-system can be reduced with the new methods. The results are given at the end of this appendix. As is to be expected in this example, a large part of the transitions can be blocked. The problem that remains is how to convert the labeled transition system, that remains after the application of the reduction methods, back to a finite state machine which can be used as an input for the existing test generation techniques. This conversion is a difficult problem.

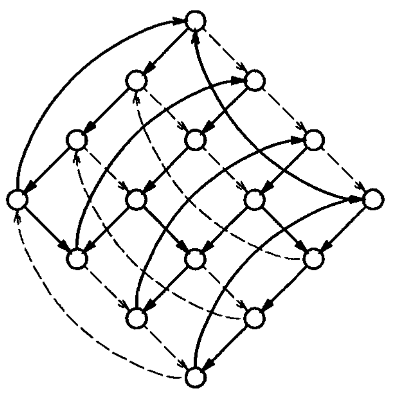


Figure D.7: Reduced labeled transition system of two completely independent processes. Reduction was done using the reduced search algorithm.

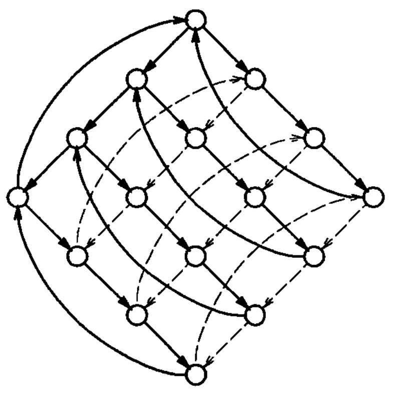
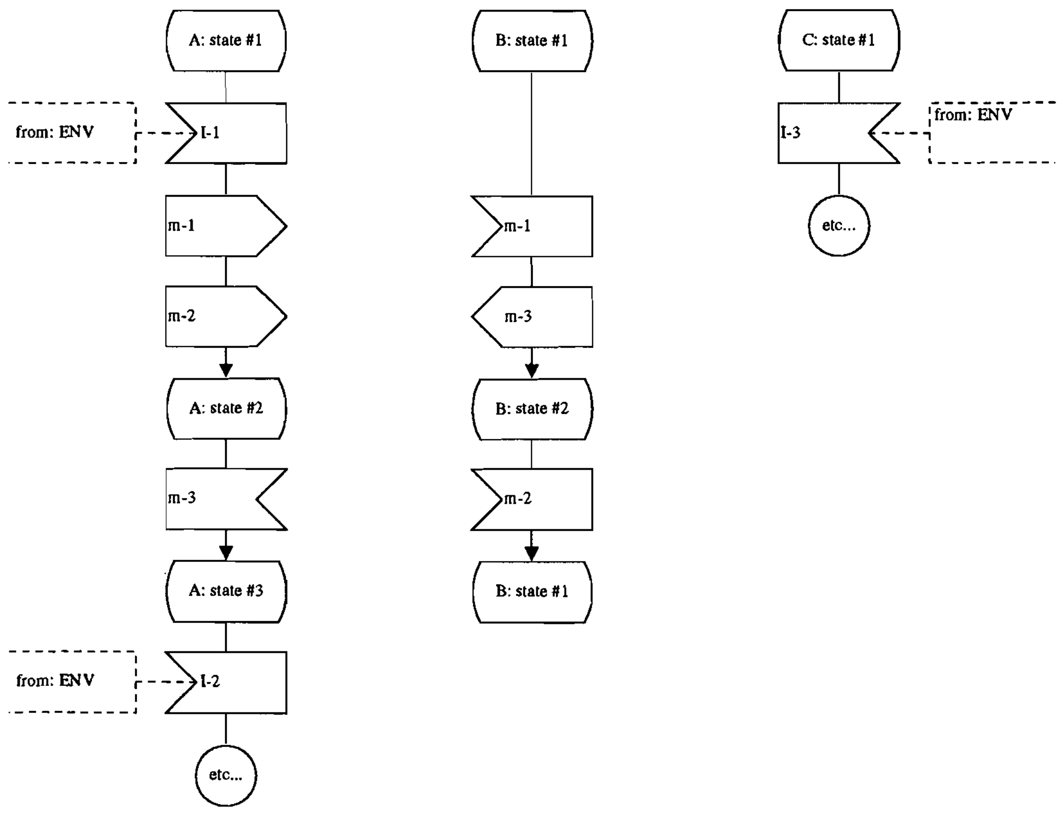


Figure D.8: Reduced labeled transition system of two completely independent processes. Reduction was done using the sleep sets algorithm.

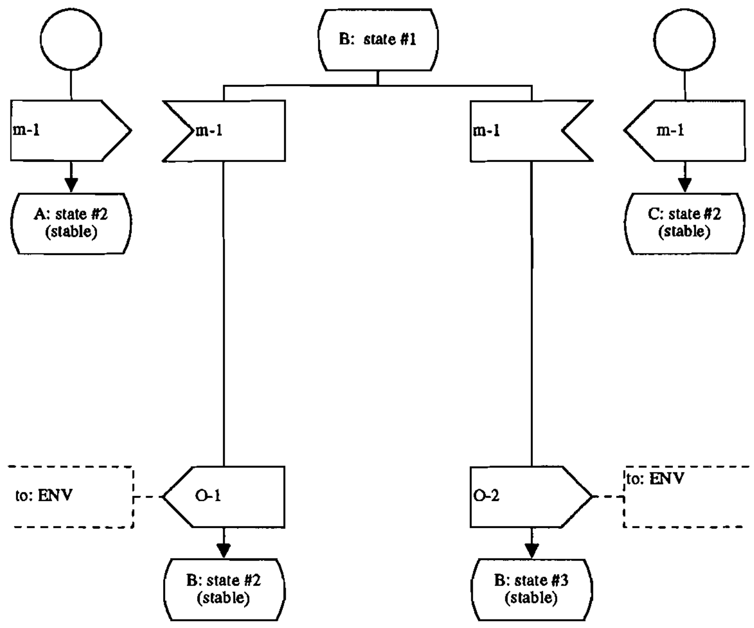
Appendix E

Examples stable traces

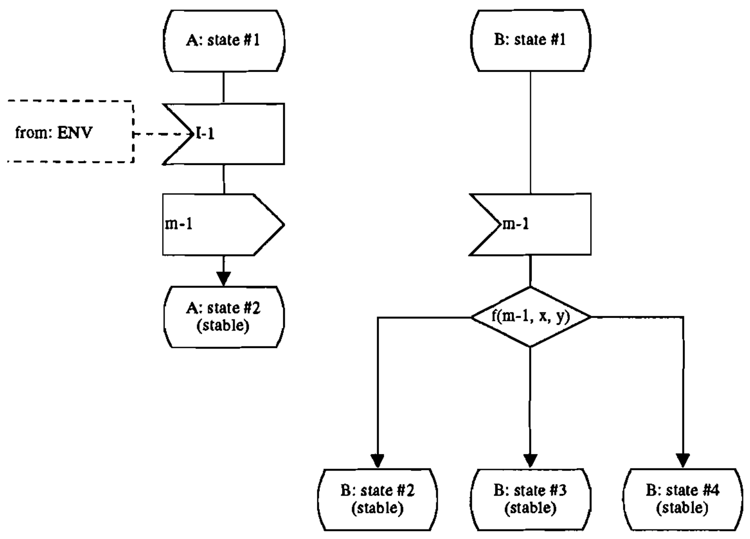
Example part of a system. In a composite state (A: state #1, B: state #1 and C: state #1), two external inputs are enabled. For input I-1, different traces exist. Every trace leads to the same global end-state (A: state #3, B: state #1 and C: state #1).



Example part of a system. During the transition from a stable composite state to a new stable composite state, process B can receive messages (m-1) and (m-2). Each message leads to a different end-state. The choice can be detected at the system boundary by the generated outputs.



Example part of a system. The composite state (A: state #1, B: state #1) is stable. The system transfers to a new stable composite state after input I-1. In the new stable state, process B resides in one of the states {#2, #3, #4}. The choice is made on the basis of parameters that are not visible in the model. The tester can not determine which state is reached ater the executionof the stable trace.


A: state #1
from: ENV
I-1
m-1
A: state #2 (stable)
B: state #1
m-1
f(m-1, x, y)
B: state #2 (stable)
B: state #3 (stable)
B: state #4 (stable)