An Integrated Approach to Testing Complex Systems

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Universität Dortmund
am Fachbereich Informatik

von
Oliver Niese

Dortmund
2003

Tag der mündlichen Prüfung: 12.12.2003

Dekan:	Prof. Dr. Bernhard Steffen
Gutachter:	Prof. Dr. Bernhard Steffen
	Prof. Dr. Peter Buchholz

An Integrated Approach to Testing Complex Systems

Oliver Niese

August 4, 2003

To
Ilona Niese

Abstract

The increasing complexity of today's testing scenarios for complex systems demands an integrated, open, and flexible approach to support the management of the overall test process. "Classical" model-based testing approaches, where a complete and precise formal specification serves as a reference for automatic test generation, are often impractical. Reasons are, on the one hand, the absence of a suitable formal specification. As complex systems are composed of several components, either hardware or software, often pre-built and third party, it is unrealistic to assume that a formal specification exists a priori. On the other hand, a sophisticated test execution environment is needed that can handle distributed test cases. This is because the test actions and observations can take place on different subsystems of the overall system.

This thesis presents a novel approach to the integrated testing of complex systems. Our approach offers a coarse grained test environment, realized in terms of a component-based test design on top of a library of elementary but intuitively understandable test case fragments. The relations between the fragments are treated orthogonally, delivering a test design and execution environment enhanced by means of light-weight formal verification methods. In this way we are able to shift the test design issues from total experts of the system and the used test tools to experts of the system's logic only. We illustrate the practical usability of our approach by means of industrial case studies in two different application domains: *Computer Telephony Integrated* solutions and *Web-based* applications.

As an enhancement of our integrated test approach we provide an algorithm for generating approximate models for complex systems a posteriori. This is done by optimizing a standard machine learning algorithm according to domain-specific structural properties, i.e. properties like prefix-closeness, input-determinism, as well as independency and symmetries of events. The resulting models can never be exact, i.e. reflect the complete and correct behaviour of the considered system. Nevertheless they can be useful in practice, to represent the cumulative knowledge of the system in a consistent description.

Zusammenfassung

Die steigende Komplexität heutiger Testszenarien für komplexe Systeme erfordert einen ganzheitlichen und offenen Ansatz zur Verwaltung des gesamten Testprozesses. Eine Anwendung klassischer modellbasierter Testansätze, in denen eine präzise und vollständige formale Spezifikation des Systems als Referenz zur automatischen Testfallgenerierung dient, ist in der Praxis nicht möglich. Gründe dafür liegen zum einen im Fehlen einer adäquaten formalen Spezifikation. Komplexe Systeme sind aus verschiedenen Komponenten zusammengesetzt, teils Hardware teils Software und oft auch aus Fremdkomponenten. Dadurch ist es inhärent unrealistisch anzunehmen, dass eine solche formale Spezifikation a priori existiert. Andererseits muss eine ausgereifte Testumgebung die Ausführung von verteilten Testfällen unterstützen, denn die Test-Stimuli und -Beobachtungen können an verschiedenen Teilkomponenten des Systems stattfinden.

Diese Arbeit präsentiert einen neuartigen Ansatz für das ganzheitliche Testen komplexer Systeme. Der Ansatz stellt eine grobgranulare Testumgebung zur Verfügung, die mittels einer komponentenbasierten Testfallbeschreibung realisiert ist. Die Basis dafür bildet eine Bibliothek von elementaren, aber intuitiv verständlichen Testfallfragmenten. Die Beziehungen zwischen den Testfallfragmenten sind orthogonal. Dies ermöglicht eine Testbeschreibung und -ausführung, die durch formale Verifikationsmethoden ergänzt wird. Hierdurch können die Testfallbeschreibungsaspekte von Experten des Systems und der verwendeten Testwerkzeuge zu Experten der Systemlogik verschoben werden. Der Ansatz wird durch verschiedene, industrielle Fallstudien in zwei verschiedenen Bereichen illustriert: Computer Telephony Integrations Lösungen und Webbasierte Applikationen.

Als Erweiterung des ganzheitlichen Testansatzes wird ein Algorithmus zur a posteriori Generierung approximativer Modelle für komplexe Systeme vorgestellt. Dafür wurde ein bekannter Algorithmus aus dem Maschinellen Lernen an applikationsbedingte Charakteristika angepasst, wie Präfix-Abgeschlossenheit, Input-Determinismus, sowie Unabhängigkeit und Symmetrien zwischen Aktionen. Die resultierenden Modelle können zwar nie exakt sein, in dem Sinne, dass sie das vollständige und korrekte Systemverhalten abbilden. Dennoch können sie von hohem praktischen Nutzen sein, da sie das gesammelte Wissen über das System in einer konsistenten Beschreibungsform repräsentieren.

Acknowledgements

I would like to express gratitude to all those people who have supported me throughout my research.

First I would like to thank my supervisor Prof. Dr. Bernhard Steffen for accepting me as a Ph.D. student and introducing me to the world of formal methods. Special thanks goes for him being a source of inspiration and advising me to find the right balance between theory and practice. His infectious enthusiasm is always a great motivation.

I am grateful to Prof. Dr. Peter Buchholz for his careful review of this work.

Furthermore, I would like to thank my former employer Dr. Tiziana Margaria, for giving me the opportunity to work on the *ITE* project, which was the foundation of this thesis, and many fruitful discussions.

This thesis won't have been written without the support of my former colleagues Dr. Volker Braun, Prof. Dr. Andreas Hagerer, and Dr. habil. Hardi Hungar. They have participated in many inspiring discussions and also provided valuable comments on early versions of this thesis, which I grateful acknowledge.

I thank all members of the former *ITE* team, particulary Markus Nagelmann and my former students, Markus Bajohr, Christian Fischbach, Harald Raffelt, and Andrea Schweer for implementing parts of the *ITE*. It has been a great pleasure to work with my colleagues at Siemens, Dr. Georg Brune, Dr. Hans-Dieter Ide, Werner Goerigk, Andrei Erochok, and Bernhard Hammelmann, who introduced me into the depths of telecommunication systems. Klaus Kolodziejczyk-Strunck supported me with interesting discussions about protocol details.

I also participated from many discussions with my current colleagues Dr. habil. Markus Müller-Olm, Claudia Gsottberger, and Haiseung Yoo.

Linda Offenburger helped polishing my English, however, all remaining mistakes in this thesis are my very own and she is not to blame at all.

Last but not least, I would like to take this opportunity to thank my family and friends for their constant moral support, and particulary my wife Melanie and my son Cornelius for their love, for sharing ups and downs, and for reminding me, when necessary, that computer science is not the most important thing in my life.

Contents

List of Figures

List of Tables

Part I

General

Chapter 1

Introduction

This thesis presents a novel approach to the integrated testing of complex systems. Its development is driven by the observation that "classical" formal-method-based approaches fail to enter in practice. Therefore, our approach offers a coarse-grained test environment, realized in terms of a component-based test design on top of a library of elementary but intuitively understandable test case fragments. The relations between the fragments are treated orthogonally, delivering a test design and execution environment enhanced by means of light-weight formal verification methods. In this way we are able to shift the test design issues from total experts of the system and the used test tools to experts of the system's logic only.

This chapter gives a general introduction to the subject, motivations, aims and scientific contributions of our work.

1.1 Background and Motivation

A common trend in system development nowadays is the construction of so-called complex systems, i.e. systems consisting of several components, either hardware or software, often pre-built and third-party. Typical examples for complex systems are composite systems, like *Computer Telephony Integration* (*CTI*) solutions. Here whole hardware/software solutions, composed themselves out of special computer systems equipped with telephony hardware and corresponding software, are connected to a telephone switch. Other examples are distributed software architectures, like *Web-based* applications, where software running on a web server interacts with several clients.

Our starting point was the testing of complex *Computer Telephony Integration* solutions in a project together with Siemens AG [Sie]. Here we were faced with the

problem of performing functional regression testing for a legacy telephony switch in cooperation with over 200 *value-added* applications, to assure that they were compliant to the switch. As those applications communicate with the telephone switch via standardized protocols, namely the *Computer Supported Telecommunication Applications* (CSTA) protocol and the *Telephony Application Programming Interface* (TAPI), they are developed mainly by third-party vendors. Furthermore, the software of the telephone switch is changing rapidly, i.e. 4 major releases per year. This implies that after every release of the telephone switch all of the 200 *value-added* applications have to be recertified, often in various combinations as well. Obviously this task demands test automation. Unfortunately "classical" formal-methods based approaches, i.e. where test suites are generated with respect to a formal specification of the system, are not applicable in this situation. The reasons are on the one hand that a formal specification of the sort of systems described above is not available, because parts of the overall system are third-party. But even for the telephone switch a complete and precise formal specification cannot be achieved for both complexity reasons (a precise model will be "too large" to construct) and economic reasons (the generation and maintenance of a precise model will be "too costly") [1].
On the other hand for test execution an environment is needed that can handle distributed tests, i.e. tests where some actions/observations take place on the telephone switch and others on the application(s). But this is not covered by "classical" formal-method-based approaches. Thus a tool-supported test approach is needed that supports:

— the intuitive (manual) construction of test cases out of reusable building-blocks;

— guidance during the test case construction;

— automatic execution of distributed test cases;

— support for test run evaluation; and

— easy integration of new test scenarios.

To sum up, the increasing complexity of today's testing scenarios for complex systems demands an integrated, open, and flexible approach to support the management of the overall test process. "Classical" model-based testing approaches, where a complete and precise formal specification serves as a reference for automatic test generation, have often proved impractical. Reasons are on the one hand the absence of a suitable formal specification. Because of the characteristics of complex systems

[1] Note that we present a promising approach for the a posteriori generation of approximate models in this thesis.

it is unrealistic to assume that a formal specification exists a priori. On the other hand, a sophisticated test execution environment that can handle distributed test cases is needed. This is because the test actions and observations can take place on different subsystems of the overall system.

1.2 Main Contributions

In this thesis an integrated approach is developed for testing complex systems. The main contributions of this thesis are the design of the general approach, the development of a suitable test environment for supporting the approach, and an operational procedure that bridges the gap between formal testing theory and current testing practice by a posteriori model generation.

Integrated Test Approach

An integrated test approach for testing complex systems has been developed, in which we aim at test automation by supporting test engineers during their manual design of tests. The tests can be instantly executed within a test environment. As system-level testing usually treats the system under test from an end-user's point of view, this should be maintained when moving to an automated test execution, meaning that the test design should also happen at this level of expertise and intuition. Furthermore, the test design is accompanied by formal methods as much as possible, i.e. rules concerning the construction of "correct" tests guide test engineers during the design of tests. The formulation of the consistency rules is supported by a pattern-based approach.

I was involved in the development of the general concepts and was responsible for its realization and the formal foundation of the test approach as well as the pattern-based approach.

Integrated Test Environment

A test environment has been developed to facilitate the proposed integrated test approach. Our environment is characterized on the one hand by its open and flexible architecture so that diverse test tools can be integrated as required, where test tools carry out the test actions/observations. One the other hand, it is built on top of a CORBA-based communication layer that supports distributed test execution on heterogenous platforms. In addition a well-defined and tool-supported integration process for test tools is presented.

I was responsible for the design and implementation of the integrated test environment and advised the case studies, in particular:

— Design of the communication layer for the communication with the test tools;

— definition of the test tool integration process;

— integration of the test tool *Rational Robot*.

A Posteriori Model Generation

We present an approach for generating models for complex systems a posteriori by adapting a machine learning algorithm. In this way we are able to represent the cumulative knowledge about the system in a consistent description. Furthermore, various pieces of information about the considered system domain helps to optimize the algorithm, e.g. independency and symmetries of events.

By optimizing a standard learning method according to domain-specific structural properties, we are aiming at generating approximate models for complex reactive systems in practice. Here we considered properties like prefix-closeness, input-determinism, as well as independency and symmetries of events.

Learning is only feasible if one can check actively whether a given abstract sequence corresponds to (is an abstraction of) a concrete system behaviour. In fact it was the *integrated test environment* that enabled us to implement learning procedures in practice by bridging the gap between the abstract models and the "real" world. Its flexible test specification formalism supports the generation of concrete test cases from abstract propositional sequences. Furthermore, its precise test execution semantics ensures that we can perform the mapping of the results of the test runs back into the abstract sequences so that they can be processed by the learning algorithm. We emphasize the practicability of our method by means of experiments we have carried out in an industrial setting.

I worked on the optimizations of the standard learning procedure and additionally I adapted the learning procedure to deal with input/output systems directly. Furthermore, I implemented the algorithms and established an environment for "real" world experimentation with the help of the integrated test environment.

1.3 Related Work

The contributions of this thesis are in two different research areas: Testing and machine learning.

Testing

To our knowledge there exists neither commercial nor academic tool-supported test approaches providing comprehensive support for the whole system-level test process. In particular, most research on test automation concentrates on the generation of test cases and test suites on the basis of a formal model of the system. In the *TorX* approach [TB99] a (possibly infinite) test suite is derived from a formal model, described as a labelled transition system. Consequently this approach will usually be applied on-the-fly. In contrast the *TGV* approach [FJJ$^+$97] derives single test cases out of a formal specification – again described by a labelled transition system – with respect to a given test purpose (provided by a special sort of labelled transition systems). Test cases can be derived by computing the intersection between the model and the test purpose. The project *Agedis* [Age] tries to bring this approach into an industrial context. Furthermore, there exist several approaches based on *Standard Description Language* (*SDL*) specifications, e.g. *Autolink* [SKGH98] to name an academic one and *Telelogic Tau* [Tel] as a representative for commercial SDL test tools. These approaches usually derive test cases with respect to test purposes given by *Message Sequence Chart* (*MSC*) descriptions. Common to all of these tools is that they on the one hand presuppose the existence of a precise and finely granular formal specification in the particular formalism. On the other hand they do not care about the test execution, i.e. for the resulting test cases an execution environment has to be implemented.

Other approaches concentrate on providing (generic) test execution environments. A remarkable one is the *Test Environment Toolkit* (*TET*) [The], an extensible framework for both local or distributed testing. Test engineers develop test interfaces that can be used during the formulation of test cases. The major drawback of TET is that test cases have to be "implemented", i.e. are small programs written in the particular programming language of the application being tested. This requires a deep understanding of the system's internals in contrast to our approach, where a test engineer only needs to be expertise of the application's logic. Furthermore, no well-defined and tool-supported integration process for test tools exist. Finally, during distributed testing the involved components communicate via proprietary interfaces instead of using standard communication mechanisms like CORBA.
Another interesting, but yet new and incomplete, approach is the *Eclipse/Hyades* project [Hya]. Here software test tools can be integrated via a plug-in approach and operate on a common data model. However, the global coordination of these test tools during test execution has to be implemented "hard-coded", e.g. directly as a Java program.

Another recent upcoming approach is based on the new revision of the *Test and Test Control Notation* (*TTCN*-3) [Eur03], called *TT Series* [Tes]. One big advantage is that TTCN-3 is a standardized notation. So far various test tools already supports TTCN-2, the predecessor of TTCN-3, for exporting their test case descriptions. The *TT Series*, however, does not support a uniform handling of the involved test tools, as test cases are compiled in Java code and have to be integrated into a “self-made” and application-specific test execution environment.

To sum up, none of the approaches presented above covers all required aspects as postulated in Section 1.1.

Machine Learning

Machine learning is a wide area of active research. The idea of combining learning and testing techniques is quite new. At the theoretical level the work of [PVY99, GPY02] is a notable exception, where the problem of learning and refining system models is studied. The ultimate goal may be testing a given system for a specific property, or correcting a preliminary or invalidated model. In our approach, however, we focus on the very practical aspects of learning models of real-life systems. We improve the learning efficiency by optimizing a standard learning algorithm with respect to structural properties of the considered system domain. To our knowledge no comparable approach exists.

1.4 Publications

The research for this thesis lead to 11 publications of the intermediate scientific results and 3 (refereed) tool-related papers.

Testing of Complex Systems Our general approach for testing complex systems has been described in [NMN+00, NSM+01].

Testing of Computer Telephony Integrated Systems In [NMH+00] we have discussed how to apply our approach to the testing of *Computer Telephony Integrated* solutions. In addition in [NMH+01, HMN+01] we have shown the efficiency of our approach along industrial case studies. Finally in [MNSE02, MNS02b] we have demonstrated that our approach scales to modern IP-based telecommunication solutions. A tool presentation was given in [NNH+01].

Testing of Web-based Applications We have extended our approach to cope with *Web-based* applications as well, which is discussed in [NMS02] and a corresponding tool presentation in [MNS02a].

A Posteriori Model Generation In [HHNS02, HHM+02b] we have developed a novel way of generating models for complex systems a posteriori and in [HHM+02a] a tool presentation can be found. Furthermore, in [HNS03] we have presented an optimized learning algorithm that allows the efficient learning of such models.

1.5 Organization of this Thesis

This thesis elaborates an integrated approach for testing complex systems. It is structured into five parts:

***Part I**.* After this introductory chapter we give in Chapter 2 a brief overview of formal testing theory. We present basic terms and notations, the international standard of *Formal Methods in Conformance Testing*, and a concrete instantiation of this standard.

***Part II**.* This part introduces our integrated test approach together with a corresponding test environment for supporting the test process. In Chapter 3 the general concepts of our approach are presented and its formal foundation in particular. A concrete test environment that supports our approach is developed in Chapter 4. Finally in Chapter 5 we illustrate the usage of our test environment by means of a concrete example.

***Part III**.* In the third part we discus practical experience in industrial case studies, where our approach has been applied to the test in the area of *Computer Telephony Integrated* solutions and *Web-based* applications.

The first case study deals with the test of 11 different *Computer Telephony Integrated* solutions which we have performed with our industrial partners at *Siemens AG* [Sie]. The work on this study and its results are presented in Chapter 6.

The second case study was done on *Web-based* applications in cooperation with *METAFrame Technologies GmbH* [MET] and is presented in Chapter 7.

***Part IV**.* In the fourth part we suggest and discuss various improvement opportunities for our approach. In particular the a posteriori generation of models for complex systems is presented in Chapter 8. Here we present, in addition to the method itself, practical results that we have achieved by generating models for non-trivial telecommunication systems. Finally in Chapter 9 we draw some conclusions and provide an outlook for future work.

Part V. provides the appendices.

Chapter 2

Foundations of Testing Theory

To assure that system behaviour is not faulty which might cause severe damage, the system has to be checked to see if it behaves as expected. This process is called *validation*. For the validation of a system the desired behaviour must be known. A description of the desired behaviour is called a *specification*, which identifies what a system must do. A specification, however, does not prescribe how this is done. A system that is supposed to implement the desired behaviour is called an *implementation*. Usually it is a real, executing, system.

Two complementary validation techniques that can be used to increase the level of confidence in the correct functioning of systems as prescribed by their specifications, are *testing* and *verification*. Whereas verification aims at proving properties about the system based on a mathematical model of the system, testing is performed by experimenting with the real, executing implementation (or an executable simulation model) in order to find errors. Testing can be understood as the process of evaluating a system or its components by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results [ANS83].

Verification can ascertain whether a required property has been satisfied, but this certainty only applies to the model of the system. This implies that any verification is only as good as the validity of the system model. Testing, in practice based on observing only a small subset of all possible instances of system behaviour, is usually incomplete: *testing can show the presence of errors, not their absence*. Since testing can be applied to the real implementation, it is useful in those cases when a valid and reliable model is not present.

The outline of this chapter is as follows: first in Section 2.1 the general terms and notions of testing theory are presented. After that in Section 2.2 a framework for testing based on formal methods, "*Formal Methods in Conformance Testing*", is introduced. Finally, a concrete instance of this framework is discussed in Section 2.3.

2.1 Introduction to Testing Theory

Testing can take place at different levels of abstraction:

— *Unit Testing:* Testing the smallest testable piece of software or hardware

— *Component Testing:* Testing the cooperation of a number of units that make up a component

— *System-level Testing:* Testing the complete system

For each of these levels of abstraction different aspects of a system can be tested:

— *Functional/Conformance Testing:* Tests whether the behaviour of the implementation conforms to the specified behaviour. Its primary objective is to assess whether the system under test is conforming to end-user requirements.

— *Regression Testing:* Determines whether any errors have been introduced during the error-fixing process. Note that it is in this area of regression testing that automated test tools offer the largest return on investment.

— *Stress Testing:* Testing the performance under heavy workload. It measures the capacity and resiliency of the system, often on several hardware platforms. The system is asked to process a huge amount of data or perform many function calls within a short period of time. A typical example could be to perform the same function call from all workstations simultaneously on the server.

— *Performance Testing:* In performance testing we are interested in verifying that the system under test meets specific performance efficiency objectives. Performance testing can measure and report on such data as input/output rates, average database query response time, and CPU utilization rates. The same tools used in stress testing can generally be used in performance testing.

— *Robustness Testing:* Tests how an implementation reacts to unspecified, or "abnormal" environments.

— *Acceptance Testing:* Tests whether the initial system requirements are met when the system operates in its intended production environment.

To test an implementation, called *system under test*, a tester must access the implementation and carry out tests against this implementation. Basically three different levels of accessibility are distinguished and these corresponds to different types of testing: *white-box testing*, *grey-box testing*, and *black-box testing*. If the internal structure of an implementation is fully known, then this information can be utilized when the implementation is tested. This is called white-box testing. An example of white-box (software) testing is when the implementation consists of a piece of code, and the source code of the implementation is available to the tester; in that case the tester can access, and measure, specific internal characteristics of the implementation (e.g., if particular statements are executed, or if certain variables contain the expected values or not). The opposite of white-box testing is black-box testing. In black-box testing it is assumed that the implementation can only be accessed through its interface with the environment, and no knowledge of the internal structure of the implementation is present. Black-box testing is done when, for example, the source code of an implementation is not available but only the executable is, or in case of third-party testing. Grey-box testing lies in between black-box testing and white-box testing. In grey-box testing it is assumed that only part of the internal structure of an implementation is known.

In this thesis we concentrate on functional testing at system-level, which is often used as a synonym for black-box testing, because during system-level testing the test purposes deal mostly with the system's externals. Functional testing is concerned with checking implementations against their specifications by means of experimentation. Tests are derived from the (informal) specification, then applied to the implementation under test, and, based on observations made during the execution of the tests, a verdict about the correct functioning of the implementation is given. Within system-level testing this is a mainly manual, laborious and time-consuming process, as usually, neither a complete and precise specification of the overall system exists, nor is there an execution environment for the test execution.

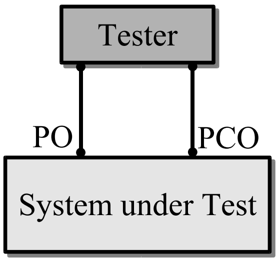


Figure 2.1: Simple Test Architecture

A *test architecture* is an environment in which a system is tested, cf. Figure 2.1. It describes at a high level of abstraction how the tester communicates with the system under test. It consists of a tester and a system under test, which provides access via *points of control and observation* (*PCO*) and offers observation interfaces via *points of observation* (*PO*). The PCO's and PO's define the *test interface* between the tester and the system under test. Note that often an additional indirection step is introduced through a special *test context*, as in practice not all system under test offer PCO resp. PO. In such cases the test context is responsible for providing the test interfaces.

2.2 Formal Methods in Conformance Testing

The standard '*Formal Methods in Conformance Testing*' (FMCT) defines a framework for the use of formal methods in conformance testing [ISO96] and is complementary to the international standard *IS-9646* '*OSI Conformance Testing Methodology and Framework*' (CTMF) [ISO91], that is mainly intended for specifications written in a natural language. FMCT is intended to guide the testing process of an implementation with respect to a formal specification, and it defines, at a high level of abstraction, the concepts used in conformance testing, such as conformance, testing, test generation, etc. In this section the main concepts of FMCT about conformance, testing, and conformance testing, are recalled. For a detailed description we refer to the FMCT document itself [ISO96].

2.2.1 Specifications and Implementations

Within FMCT it is assumed that specifications prescribe the behaviour of a system by means of *formal description techniques* (*FDT*). A specification s is therefore a formal object, contained in the set of specifications $SPECS$, that can be expressed by means of the particular FDT. The implementations are real systems and not mathematical objects, thus making it impossible to define a formal relation between specifications and implementations. By a *test assumption* or *test hypothesis* we suppose that any implementation $IUT \in IMPS$, where $IMPS$ denotes the universe of implementations, can be modelled by a formal model $m_{IUT} \in MODS$, where $MODS$ represents the universe of models of a particular formalism (e.g. labelled transition system, finite state machines, etc.). In order to facilitate the comparison, the formalism $MODS$ may be chosen to be the same as $SPEC$. Note that the test assumption only assumes that such a model exists, and not that this model is known a priori.

2.2.2 Conformance

There are two approaches to define conformance. One depends on the abstract concept of the implementation relation, the other one relates to the more concrete concept of conformance requirements. The latter can be seen as a refinement of the former definition.

Implementation Relation

The conformance between specification and implementation is formally characterized by a relation between the model m_{IUT} and the specification s and is called an *implementation relation*:

$$\mathbf{imp} \subseteq MODS \times SPECS$$

The implementation relation is not determined by the combination of $MODS$ and $SPECS$, as there can exist several different implementation relations for a given formalism.

There may also be more than one implementation conforming to a specification. For a specification $s \in SPECS$ and an implementation relation **imp** there exists a set M_s^{imp}, denoting the set of all conforming models $m \in MODS$, with respect to s and **imp**:

$$M_s^{imp} = \{m \in MODS \mid m \ \mathbf{imp}\ s\}$$

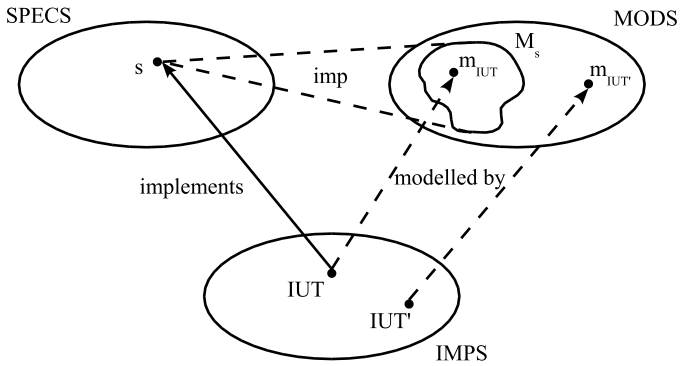


Figure 2.2: Relations between Specifications, Implementations, and their Models

The relations between specifications, implementations and their models are depicted in Figure 2.2. The implementation IUT is correctly implementing the specification s if the implementation IUT may be modelled by the model m_{IUT} and if this model in turn is part of the set of models M_s that implement the specification s according to the implementation relation **imp**. It is important to note that the conformance between an implementation and a specification depends on the chosen implementation relation. For the formalism of labelled transition systems or input/output automata there exists a variety of different implementation relations. Usually *failure* or *testing* equivalences or preorders are defined in terms of tests whose processes or models may or must satisfy [DNH84, Hen85, Hoa85], and also the work of [CC92, CS90, Hee98, Lan90, NC95, Seg93, Tre92, Tre96a, VT95]. There also exists variants where bisimulations ([Mil89, Par81, Wal88]) are used rather than failure relations, e.g. [Abr87, CH93].

Conformance Requirements

The second definition of conformance is based on the concept of conformance requirements as used in CTMF. These requirements denote properties to be satisfied by the (model of the) implementation that claims to conform to the specification.

Let $REQS$ be the set of all requirements which may be expressed in a particular formal requirement language. A specification $s \in SPECS$ may now be expressed as a set of requirements $R_s \subseteq REQS$, and the set of all possible specifications may be defined as $SPECS = \mathcal{P}(REQS)$, i.e. as the powerset of the possible requirements. The set R_s is called a *requirement specification*, and a single element of R_s is called a *requirement*.

An implementation modelled by m_{IUT} conforms to a specification s if all requirements $r \in R_s$ are satisfied by m_{IUT}. Thus the relation **sat** is defined to be:

$$\mathbf{sat} \subseteq MODS \times REQS$$

The set M_s of models conforming to the specification s may then be defined as

$$M_s = \{m \in MODS \mid \forall r \in R_s : m \ \mathbf{sat} \ r\}$$

An implementation IUT conforms to a specification s, if it can be modelled by a model $m_{IUT} \in M_s$, since M_s is the set of all models that satisfy all conformance requirements R_s of the specification s. In other words, an implementation IUT conforms to a requirement specification R_s if its model m_{IUT} satisfies at least all requirements in R_s, and possibly more.

2.2.3 Testing

Testing relates to the experiment where an IUT is stimulated and its responses are observed. This experiment is called *test execution* and is performed within a concrete test architecture. The test execution procedure is specified by *test cases* $t \in T$, where t represents a single test case and T denotes a set of test cases, called *test suite*. The test cases are specified by means of a test notation referred to as $TESTS$, i.e., $t \in T \subseteq TESTS$. During test execution observations o out of the domain of the observations OBS are made. To each observation there is assigned a *verdict*, denoted by the function

$$\mathbf{verd}_t : OBS \rightarrow \{pass, fail\}$$

which must exist for all test cases $t \in T$. Test execution may be formalized by:

$$\mathbf{exec} : TESTS \times MODS \rightarrow OBS$$

Thus $\mathbf{exec}(t, m_{IUT})$ calculates the observations resulting from the application of test case t to the model m_{IUT}. By means of this function we may express the set of models P_t that pass a test case t with:

$$P_t = \{m \in MODS \mid \mathbf{verd}_t(\mathbf{exec}(t, m)) = pass\}$$

Based on the set P_t for test case t we can state that:

$$IUT \ \mathbf{passes} \ t \Leftrightarrow m_{IUT} \in P_t$$

Applying it to test suites as a whole we may say that an IUT passes a test suite T if the *pass* verdict is assigned to all executions of test cases $t \in T$:

$$IUT \ \mathbf{passes} \ T \Leftrightarrow m_{IUT} \in P_T, \text{ where } P_T = \bigcap_{t \in T} P_t$$

2.2.4 Conformance Testing

In conformance testing we link the concepts of the implementation relation and testing. What we need is a test suite T for which we can say: an IUT passes that test suite if (and only if) the IUT conforms to the specification with respect to the implementation relation **imp**:

$$m_{IUT} \ \mathbf{imp} \ s \Leftrightarrow IUT \ \mathbf{passes} \ T$$

Such a test suite is called *complete*, but in general it is not possible to create test suites that allow the implication in both directions: these test suites are usually infinite and consequently cannot be carried out in practice. A test suite can be

Exhaustive The set of implementation models that pass T is a subset of the models conforming to the specification s, i.e., $P_T \subseteq M_s$ (implication '$\Leftarrow$' in the last equation),

Sound The set of implementation models that pass T is a superset of the models conforming to the specification s, i.e., $P_T \supseteq M_s$ (implication '$\Rightarrow$'), or

Complete The set of implementation models that pass T is equal to the set of the models conforming to the specification s, i.e., $P_T = M_s$ (implication '$\Leftrightarrow$').

There exists a variety of different test generation algorithms, which can generate test suites out of a given formal specification, e.g. labelled transition systems or (extended) finite state machines. Some test generation methods focus on the derivation of complete test suites rather than on computing finite test suites, e.g. [BSS86, Bri88, Tre92, Tre96a]. Other approaches generate test cases with respect to certain formal test purposes, e.g. as can be found in the TGV approach [FJJ$^+$96, FJJ$^+$97]. In the domain of finite state machines several test generation methods exist, where finite test suites with respect to given fault models are generated, beginning from a simple state coverage up to a sophisticated state verification, e.g. [Vas73, Cho78, CVI89, EP90, FvBK$^+$91, ADLU91, LPvB93, vBP94, LvBP94, PvBY96]. These methods were also adapted for dealing with labelled transition systems resp. input/output automata, see [TP98, TPvB96]. There are, however, a lot of other approaches, e.g. [NT81, SD88, SL88, UZ93, Ber94, PYvBD96, SKGH98, YCL98, RNHW98, RS98, PW99, God99a, KSK00, FHP02]. Please refer to [LY96, BT00] for good surveys of the different test generation approaches.

2.3 Instantiation of the FMCT Framework

By instantiating the abstract concepts defined in the previous section with specific choices, the formal conformance testing framework can be used in practice.

2.3.1 Models

The formalism of labelled transition systems is used to describe the behaviour of a system.

Definition 2.1 (Labelled Transition System). A labelled transition system ($\mathcal{LTS}$) is a structure $\mathcal{S} = (\Sigma, A, \rightarrow, s_0)$, consisting of a countable, non-empty set Σ of states, a countable set A of labels or actions, a transition relation $\rightarrow \subseteq \Sigma \times A \cup \{\tau\} \times \Sigma$, and a unique start state s_0.
The class of all labelled transition systems over A is denoted by $\mathcal{LTS}(A)$.

The actions in A represent the external observable behaviour of the system, where the special action τ denotes an unobservable, internal behaviour.

Let $S = (\Sigma, A, \rightarrow, s_0)$ be a labelled transition system with $s, s' \in \Sigma$, $\mu_{(i)} \in A \cup \{\tau\}$, $a_{(i)} \in A$ and $\sigma \in A^*$, then the following notations can be defined:

$$
\begin{array}{lll}
s \xrightarrow{\mu} s' & =_{def} & (s, \mu, s') \in \rightarrow \\
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' & =_{def} & \exists s_1, \ldots, s_{n+1} : s = s_1 \xrightarrow{\mu_1} s_2 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s_{n+1} = s' \\
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & =_{def} & \exists s' : s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' \\
s \overset{\epsilon}{\Longrightarrow} s' & =_{def} & s = s' \text{ or } s \xrightarrow{\tau \cdot \ldots \cdot \tau} s' \\
s \overset{a}{\Longrightarrow} s' & =_{def} & \exists s_1, s_2 : s \overset{\epsilon}{\Longrightarrow} s_1 \xrightarrow{a} s_2 \overset{\epsilon}{\Longrightarrow} s' \\
s \overset{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} s' & =_{def} & \exists s_1 \ldots s_{n+1} : s = s_1 \overset{a_1}{\Longrightarrow} s_1 \overset{a_2}{\Longrightarrow} \ldots \overset{a_n}{\Longrightarrow} s_{n+1} = s' \\
s \overset{\sigma}{\Longrightarrow} & =_{def} & \exists s' : s \overset{\sigma}{\Longrightarrow} s' \\
s \overset{\sigma}{\not\Longrightarrow} & =_{def} & \text{not } \exists s' : s \overset{\sigma}{\Longrightarrow} s'
\end{array}
$$

A sequence of observable actions is called a *trace*. The traces of a labelled transition system specification S, denoted by $Tr(S)$, are all sequences of visible actions that S can perform. If σ is a trace of S, then any initial part of σ is also a trace of S. We call this a *prefix* of σ.

Definition 2.2 (Trace, Prefix). Let $S \in \mathcal{LTS}$.

1. $Tr(S) =_{def} \{\sigma \in A^* \mid S \overset{\sigma}{\Longrightarrow}\}$
2. A trace σ_1 is a *prefix* of σ_2, denoted by $\sigma_1 \preceq \sigma_2$, if $\exists \sigma' : \sigma_1 \cdot \sigma' = \sigma_2$. Since σ' is unique we write $\sigma' = \sigma_2 \backslash \sigma_1$.

Some additional definitions can be made: these are especially important in the consideration of the conformance relations.

Definition 2.3. Let s be a (state out of a) labelled transitions system and let S be a set of states.

1. $init(s) =_{def} \{\mu \in A \cup \{\tau\} | s \xrightarrow{\mu}\}$
2. $init(S) =_{def} \bigcup \{init(s) | s \in S\}$
3. s **after** $\sigma =_{def} \{s' | s \xrightarrow{\sigma} s'\}$
4. S **after** $\sigma =_{def} \bigcup \{s \textbf{ after } \sigma | s \in S\}$

However many real-world systems can be described as reactive systems, i.e. they react to stimuli from their environment. To describe these system type the granularity of labelled transitions systems is often not fine enough. We have to distinguish between input and output actions. The formalism of *input/output automata* [LT89] provides exactly this partition of A.

Definition 2.4 (Action signature). A partition of a set A of actions into three disjoint sets of input actions (A_I), output actions (A_O) and internal actions (A_{int}) is called an *action signature*.

Proposition 2.5. Let A be an action signature, then the set $A_{ext} = A_I \cup A_O$ is called the set of *external observable actions*.

Proposition 2.6. Let A be an action signature, then the set $A_{local} = A_O \cup A_{int}$ is called the set of *locally controlled actions*.

Definition 2.7 (Input/Output Automaton). An Input/Output automaton ($\mathcal{IOA}$) is a structure $\mathcal{S} = (\Sigma, A, \rightarrow, s_0, P)$, where Σ denotes a set of states, A a set of actions, $\rightarrow$ a transition relation $\rightarrow \subseteq \Sigma \times A \times \Sigma$, s_0 is a unique start state and the following holds:

1. A is an action signature
2. P is a partition of *local*
3. $\forall s \in \Sigma, a \in A_I : s \xrightarrow{a}$

Note that an $\mathcal{IOA}$ can be seen as a special case of a $\mathcal{LTS}$, i.e. $\mathcal{IOA} \subseteq \mathcal{LTS}$.

In testing a weakened version of input/output automata will often be used: these are called *input/output labelled transition system* ($IOTS$) [Tre96a, Tre96b]. Instead of requiring each state to be input-enabled, in an $IOTS$, each input must be reachable within a transitive closure of internal actions. Note, however, that this weakening conflicts with the fairness properties of $\mathcal{IOA}$, as now infinite internal computations are possible.

The model of $\mathcal{IOA}$ serves as a formalism for describing the behaviour of both specifications and implementations. Note that sometimes the more general model of labelled transition systems is used to describe the behaviour of specifications.

2.3.2 Conformance

In the most prominent implementation relation, the testing preorder [DNH84, DN87], it is assumed that the behaviour of external observers, like the behaviour of implementations and specifications, can be modelled as labelled transition systems or input/output automata. Furthermore it is assumed that these observers communicate in a synchronous and symmetric way with the system under test. From the observer o and the system under test p, the binary infix operator $\|$ creates a labelled transition system $o \parallel p$ that models the behaviour of o experimenting on p in a synchronous way.

Definition 2.8 (Synchronous Product). Let $o, p \in \mathcal{LTS}$, then the operator $\|: \mathcal{LTS} \times \mathcal{LTS} \to \mathcal{LTS}$ is defined by the following inference rules:

1. $\dfrac{o \xrightarrow{\tau} o'}{o \parallel p \xrightarrow{\tau} o' \parallel p}$

2. $\dfrac{p \xrightarrow{\tau} p'}{o \parallel p \xrightarrow{\tau} o \parallel p'}$

3. $\dfrac{o \xrightarrow{a} o', p \xrightarrow{a} p'}{o \parallel p \xrightarrow{a} o' \parallel p'} \; (a \in A)$

Note that the third rule denotes that only observable actions that can occur are the ones agreed upon by both the observer and the system under test. If the system under test p cannot match an action offered by the observer o, then this action will not take place, and $o \parallel p$ is not able to continue, if both o and p are stable, i.e. cannot leave their current states through internal τ-actions.

We present the testing preorder $\leq_{te}$ in a slightly different way than in the original definition of [DNH84].

Definition 2.9 (Testing Preorder). The testing preorder $\leq_{te} \subseteq \mathcal{LTS} \times \mathcal{LTS}$ is defined by:
$i \leq_{te} s =_{def} \forall o \in \mathcal{LTS} : obs_{may}(o, i) \subseteq obs_{may}(o, s)$ and $obs_{must}(o, i) \subseteq obs_{must}(o, s)$
where $obs_{may}(o, p) =_{def} \{\sigma \in A^* \mid (o \parallel p) \stackrel{\sigma}{\Longrightarrow}\}$, and
$obs_{must}(o, p) =_{def} \{\sigma \in A^* \mid init((o \parallel p) \textbf{ after } \sigma) = \emptyset\}$

The testing preorder is defined in an extensional way, i.e. by comparison on the basis of external behaviour of the system under test. Intuitively an implementation i is testing preorder related to a specification s, if for every external observer o, modelled as a labelled transition system, each trace that $o \parallel i$ can perform can be performed by $o \parallel s$, and each deadlock trace of $o \parallel i$ is one of $o \parallel s$. Here a deadlock trace is a trace, after which the system is unable to perform any further action of A. Testing preorder allows implementations to be "more deterministic" than their specification, but it does not allow that implementations can do more than specified. In this sense the specification prescribes what behaviour is allowed and what is not.

2.3.3 Testing

Now that we have chosen the models for specifications and their implementations, i.e. labelled transition systems resp. input/output automata, we can define the observers, needed in Definition 2.9. We call these observers *tester processes* or *test cases*.

Definition 2.10 (Tester Process). A *tester process* $tp = (\Sigma, A, \rightarrow, s_0, P)$ for an implementation $I = (\Sigma^I, A^I, \rightarrow^I, s_0^I, P^I) \in \mathcal{IOA}$ is an $\mathcal{IOA}$ ($\mathcal{TP} \subset \mathcal{IOA}$), such that

1. $A_I = A_O^I$, $A_{Int} = \emptyset$, and $A_O \subseteq A_I^I$

2. tp is deterministic and has finite behaviour,

3. Σ contains the terminal states **pass** and **fail**, with $init(\textbf{pass}) = init(\textbf{fail}) = \emptyset$.

Definition 2.11 (Test Run). A *test run* of a test process $t \in \mathcal{TP}$ with an implementation $I \in \mathcal{IOA}$ is a computation of the synchronous product $t \parallel I$ leading to a terminal state of t:
σ is a test run of t and $I =_{def} \exists I' : t \parallel I \stackrel{\sigma}{\Longrightarrow} \textbf{pass} \parallel I'$ or $t \parallel I \stackrel{\sigma}{\Longrightarrow} \textbf{fail} \parallel I'$

Note that as tester processes have finite behaviour it is always ensured that a test run is finite. Additionally the definition of the alphabet ensures that the test process is compatible to the implementation, i.e. the outputs of the implementation are the inputs of the test process and the inputs of the implementation are the outputs of the test process. This means that an implementation can never block stimuli of the test process, and the test process is always able to process an output of the implementation.

Having this in mind an implementation i passes a test process t if all their test runs lead to the **pass**-state of t:

$$i \ \textbf{passes} \ t \ =_{def} \ \forall \sigma \in A^* : i \parallel t \stackrel{\sigma}{\Longrightarrow} i' \parallel \textbf{pass}$$

Thus an implementation passes a test suite if it passes all its tests. The aim of test generation algorithms is now to restrict the set of all possible observers or test processes, expressed by the *forall* quantifier in Definition 2.9, to a finite subset in a systematic way to ensure a certain coverage.

2.3.4 Test Generation for Input/Output Automata

As stated in the previous section a complete test suite, i.e. an implementation pass the test suite if and only if it is conform to the specification, can in general not be achieved. Therefore test generation methods are aiming at computing a finite set of tests by ensuring a certain coverage, e.g. state coverage. In this section we present a test generation method, called *HSI-method*, tailored for specifications given as $\mathcal{IOA}$, which is presented in [TP98]. Note that it bases on the work of [Pet91], who developed this method for dealing with finite state machines.

The HSI-method bases, as all its companions, on the *W-method* [Vas73, Cho78]. These methods differ basically in their state identification facilities, i.e. the verification that a given machine, with a known state diagram, is in a particular state.

In general the testing approach can be divided into two phases:

1. Transfer the implementation to all states and ensure that it is the correct state.
2. Execute every transition of the implementation and ensure that it is in the correct state afterwards.

Here each of the two phases consists of the following actions:

1. Transfer the implementation to a certain state s_i (*Transfer*) and
2. identify the reached state s_i by applying sequences, that separates it from all other states (*State Identification*).

To transfer an implementation to a certain state, it is helpful if a reliable reset can be assumed. Having this in mind we can reset the implementation first, i.e. the implementation is now in its initial state, and transfer it afterwards to a particular state. This can be formalized as a state cover set, which will be used in the first testing phase.

Definition 2.12 (State Cover Set). Let $S \in \mathcal{IOA}$ be a specification with n states. Then a *state cover set* is defined as follows:

$$SC = \bigcup_{i=1}^{n} \{\sigma \in A^*_{Ext} | s_0 \xrightarrow{\sigma} s_i\}$$

So a state cover set contains (external observable) sequences, that brings the implementation from the start state to all other states. A more sophisticated cover set enables us to ensure, that not only all states of a (correct) implementation are covered but also all transitions. This set is called transition cover set and is used for the second testing phase.

Definition 2.13 (Transition Cover Set). Let $S \in \mathcal{IOA}$ be a specification with n states and SC a corresponding state cover. Then a *transition cover set* is defined as follows:

$$TC = \{\sigma \in SC \cdot A_{Ext} | s_0 \xrightarrow{\sigma} \}$$

A transition cover set can be obtained out of a state cover set by performing after each sequence, which leads to a certain state of the implementation, all possible transitions. Note that the operator $A \cdot B$ appends each sequence of B to each sequence of A.

To define a test suite the state identification facilities have to be formulated. For this purpose we first define when two states are distinguishable.

Definition 2.14 (Distinguishable States). Let $s, s' \in \Sigma$ be two states of an $\mathcal{IOA}$. They are said to be *distinguishable*, if there exists a $\sigma \in A^*_{Ext}$, such that $init(s$ **after** $\sigma) \neq init(s'$ **after** $\sigma)$.

It turns out, that two distinguishable states are not trace equivalent. This means that for two distinguishable states, there exists a (external observable) sequence of inputs and outputs, such that it is a trace for one of the states, but it is not a trace for the other. An $\mathcal{IOA}$ which consist only of pairwise distinguishable states is called *reduced* $\mathcal{IOA}$ and in the remainder we will only consider reduced $\mathcal{IOA}$. Note, however, that every $\mathcal{IOA}$ can be transformed in a reduced one, with respect to trace equivalence.

There exists several state identification notions, e.g. *Distinguishing Sequence* [Gon70], *Characterization Set* [Vas73, Cho78], *Unique Input/Output Sequences* [SD88], *Partial Characterization Set* [FvBK+91] just to name the most prominent ones. The HSI-method, however, bases on the so called *harmonized state identifiers* [Pet91].

Definition 2.15 (Harmonized State Identifiers). The set $HSI = \{H_1, \ldots, H_n\}$ is denoted as a set of *harmonized state identifiers* for an $\mathcal{IOA}$, if

1. $H_i \subseteq A^*_{Ext} \cdot A_O$ for $1 \leq i \leq n$.
2. For each pair of different states $s_i \neq s_j$, there exists $\sigma \in prefix(H_i) \cap prefix(H_j)$ such that $\sigma \in Tr(s_i) \oplus Tr(s_j)$, where the operator $\oplus$ is denoted to be $A \oplus B = (A \setminus B) \cup (B \setminus A)$.

H_i is said to be a harmonized state identifier for state s_i. The harmonized state identifier for s_i captures the following property: for any other state s_j, there exists a sequence σ_i in $prefix(H_i)$ that distinguishes s_i from s_j and σ_i is also in

$prefix(H_j)$[1]. Note that because of the input-enableness of an $\mathcal{IOA}$, states can only be distinguished by outputs (cf. Definition 2.15.1)

The state resp. transition cover sets together with the harmonized state identifiers enables us to define the test suites for the two test phases.

In the first phase we bring the implementation to every state and verify with the corresponding harmonized state identifier that it is indeed in the correct state.

$$TS_1 = \bigcup_{i=1}^{n} SC_i \cdot H_i$$

In the second phase we execute the missing transitions, i.e. the transitions that have not been executed by the state cover set, and again verify that the implementation is in the correct state.

$$TS_2 = \bigcup_{i=1}^{n} (TC_i \setminus SC_i) \cdot H_i$$

A test suite for the HSI-method is now composed out of TS_1 and TS_2.

Definition 2.16 (Test Suite). Let $S \in \mathcal{IOA}$ be a specification with n states. Then a test suite can be computed as follows:

$$TS = TS_1 \cup TS_2 = \bigcup_{i=1}^{n} SC_i \cdot H_i \cup \bigcup_{i=1}^{n} (TC_i \setminus SC_i) \cdot H_i = \bigcup_{i=1}^{n} TC_i \cdot H_i$$

Test suites, generated with the HSI-method, consists of a set of test sequences, i.e. linear sequences of input resp. output actions.

[1]Note that we have extended the notation of $prefix$ (cf. Definition 2.2) for sets of sequences in the usual fashion.

Part II

The Integrated Test Approach – Concepts

Chapter 3

An Integrated Approach to Testing Complex Systems

The increasing complexity of today's testing scenarios for complex systems, i.e. systems composed out of a set of interacting subsystems, demands an integrated, open, and flexible approach to support the management of the overall test process. It turns out that "classical" model-based testing approaches, as presented in Chapter 2, are in general not applicable for the test of complex systems, because of the absence of a suitable formal specification. In our approach we are aiming at test automation by supporting test engineers during their manual design of tests, which can be instantly executed within a test environment. As system-level testing usually treats the system under test from an end-user's point of view, this should be maintained when moving to an automated test execution, meaning that also the test design should happen at this level of expertise and intuition. Furthermore, the test design is accompanied by formal methods as much as possible, i.e. rules concerning the construction of "correct" tests guides test engineers during the design of tests. Altogether this implies that we need an user-friendly and flexible formalism for the specification of tests, while preserving enough exactness to allow to check the tests against consistency rules and to provide a mapping of tests into the world of formal test theory to ensure a precise semantics. To sum up this way we are able to shift the test design issues from total experts of the system and the used test tools to experts of the system's logic only.

Within this chapter we first introduce complex systems in Section 3.1, before we discuss the specialities of testing complex systems in Section 3.2. After that we are designing an integrated approach to testing complex systems (Section 3.3). Finally in Section 3.4 we present a test specification formalism, together with a precise execution semantics and a suitable temporal logic for expressing consistency rules for tests.

3.1 Complex Systems

A common trend in system development nowadays is the construction of so-called complex systems, i.e. systems consisting of several components, either hardware or software, often pre-built and third-party. A complex system can be defined as a *set of interacting resp. cooperating components*. One key aspect of a complex system is that its overall behaviour is determined by the interaction of the communicating subcomponents or subsystems.

Typical examples for complex systems are composite systems, like *Computer Telephony Integration* (CTI) solutions (cf. Chapter 6). Here whole hardware/software solutions, composed themselves out of special computer systems equipped with telephony hardware and corresponding software, are connected to a telephone switch (or even to a network of those, acting in a compound as a "virtual switch"). Other examples are distributed software architectures, like *Web-based* applications (cf. Chapter 7), where software running on a web server interacts with several clients.

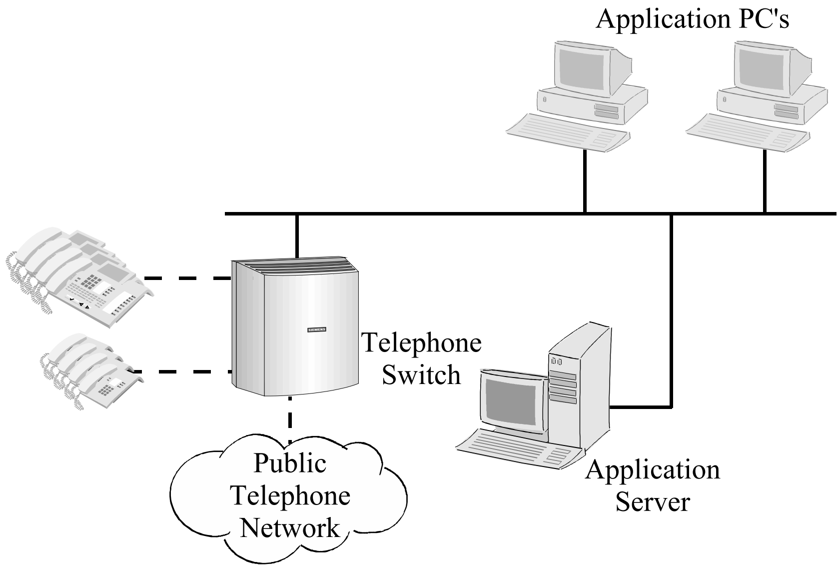


Figure 3.1: Example of a Computer Telephony Integration Solution

The (logical) architecture of a CTI solution, shown in Figure 3.1, serves as an example for a complex system. Here a telephone switch is connected to the public telephone network and operates locally connected telephones. Additionally it is connected to an application server and to several client PC's. In CTI solutions the physical telephones works in cooperation with applications located on the server resp. client PC's. So it is possible that e.g. an incoming call will be announced on a telephone and its dedicated application simultaneously or that an application can initiate a new call.

3.2 System-level Testing of Complex Systems

In general, when treating complex systems one runs into the problem that even a single subcomponent (e.g. a piece of software) depends on the underlying computer hardware (which is itself a complex system), the operating system, the device drivers, and probably several other components. In system-level testing, however, we are interested in certain aspects of the system's behaviour only, and therefore usually focus on specific components, while relying on the correctness of the others. In system-level testing the system is treated as a black-box, or at least some of its subcomponents, and one is usually interested in end-user behaviour only. For instance, when testing a CTI system one is only interested in the correct, functional interplay between a client software and a corresponding telephone and so all the aspects concerning e.g. the underlying computer hardware or the connection inbetween can be abstracted away. Note that deciding what can be seen as a single component, or subsystem, can vary, and depends inherently on the concrete testing aims.

Another aspect of testing complex systems is that it becomes increasingly unrealistic to restrict the consideration of the testing activities to the separate handling of single units of the systems only. On the contrary, since the subsystems communicate with and affect each other, scalable, integrated test methodologies are required that can handle the overall complex system as a whole. Obviously a complete complex system has to be handled on a more coarse granular level than its independent subsystems. When considering the example of Figure 3.1 one is e.g. interested in questions like: "*What happens in case of an incoming call?*" Is the information announced on both the physical telephone and on the corresponding application? To answer this question, we have to take the complete complex system into account. So the main interest in system-level testing is to ensure the (global) correct functional interplay of the involved subcomponents, rather than their intrinsic algorithmic correctness. This task is already complex enough and does not involve treating aspects like performance issues or concurrent access. Especially the concurrency can introduce non-deterministic test results, which conflicts with the regression testing situation, where repeatable results are to be achieved.

It turns out, that the simple test architecture depicted in Figure 2.1 is not sufficient to test a complex system. Instead we need to involve specialized test tools, each tailored for testing a specific aspect of the overall complex system. An example of a test architecture for the CTI solution of Figure 3.1 is shown in Figure 3.2. Here each

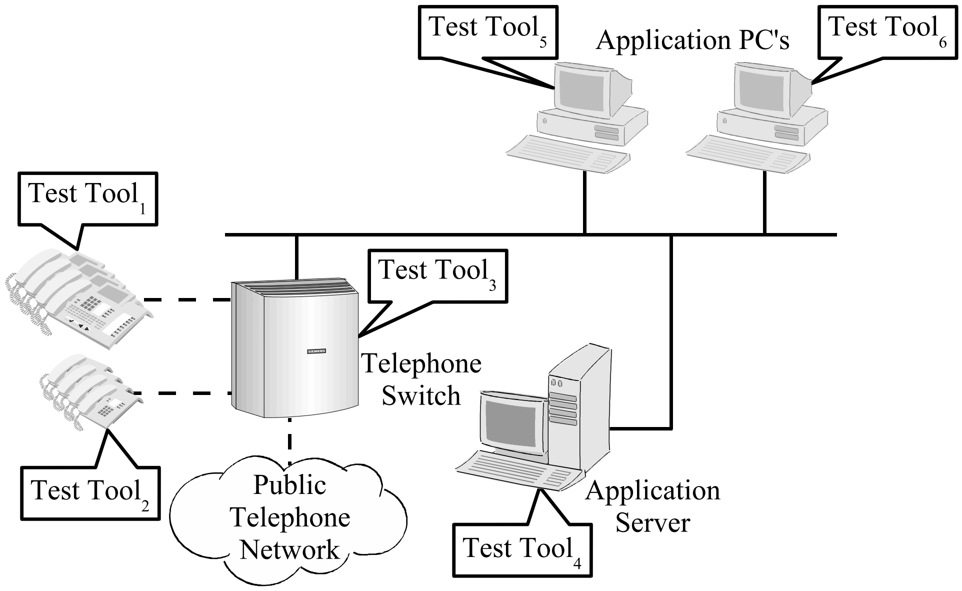


Figure 3.2: Test Architecture for a Computer Telephony Integration Solution

subcomponent, e.g. a client or a telephone, is steered and observed by its own test tool. For a system-level test, the test actions resp. observations must be distributed and coordinated throughout the different involved test tools.

3.3 Designing an Integrated Test Approach

Manual system-level testing of complex systems is a difficult and error-prone task. One reason for the complexity is the involvement of different, heterogenous test tools. So a test engineer needs to know, e.g. how to apply each of the test tools, how to evaluate the results, how to combine them (interaction), etc. Therefore a detailed technical understanding of the considered test setting is needed. Another reason for the complexity of system-level testing is that a global understanding of the overall systems's behaviour is unavoidable, e.g. how the subsystems are interacting with each other. Taken together test engineers must have knowledge in two dimensions:

— Technical overview of the used test tools and of the overall system (needed for test case execution) and

— extensive understanding of the system's logic (needed for test case design).

Furthermore for testing complex systems lots of experience is needed, because one have to take care of the interdependencies between the involved subsystems, between the subsystems and the test tools, and between the test tools itself.

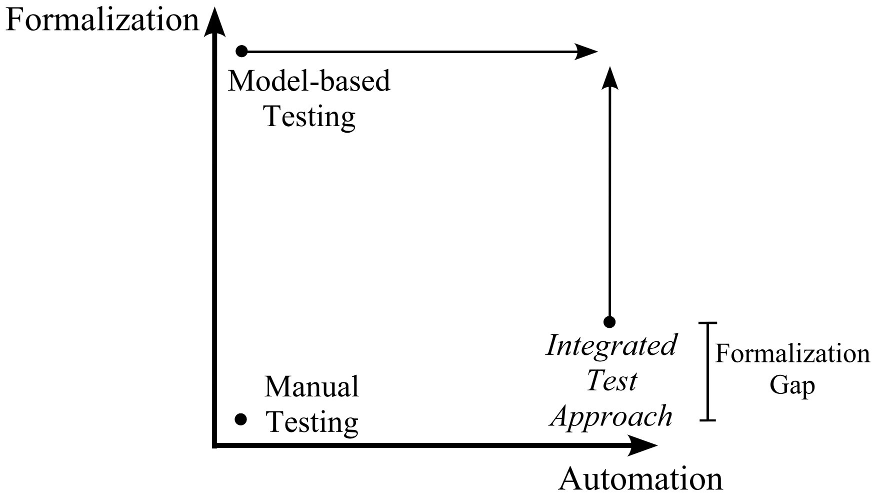


Figure 3.3: Classification of different Test Approaches

Figure 3.3 classifies three different approaches to tackle the problem of system-level testing of complex systems with respect to their degree of *Automation* and *Formalization*. With automation we denote particularly the ability of the automatic execution of tests, whereas formalization ranges from no formal specification available up to a full formal specification is available.

Manual Testing supports neither automation nor formalization, i.e. tests are executed manually and only documented informally.

Model-based Testing supports the generation of test suites out of a formal specification. For test execution, however, two non-trivial task have to be solved:

1. Implementation of a test execution environment for the resulting tests, and
2. distribution of the generic test actions among the involved test tools (see e.g. [JJKV98, TVJ00]).

Integrated Test Approach supports the manual test design in a formalism that is tailored for the automated execution within a test execution environment.

We denote the discrepancy between two approaches with respect to the degree of formalization with *Formalization Gap*. In Figure 3.3 the formalization gap between manual testing and the integrated test approach is depicted, i.e. informally documented tests vs. formalized tests. Obviously to minimize the additional effort when shifting towards test automation, the initial formalization gap should be as small as possible. When comparing the formalization gap between manual testing and model-based testing approaches resp. the integrated test approach it turns out, that, independent from a concrete formalism, the formalization gap is bigger for model-based testing. The reason is, that we need a full formal specification before we can even start the testing process, whereas in the integrated test approach we can start testing right after the specification of a single test case. The more test cases exists in the integrated test approach, possibly enriched by additional information about the considered system, the higher is the degree of formalization.

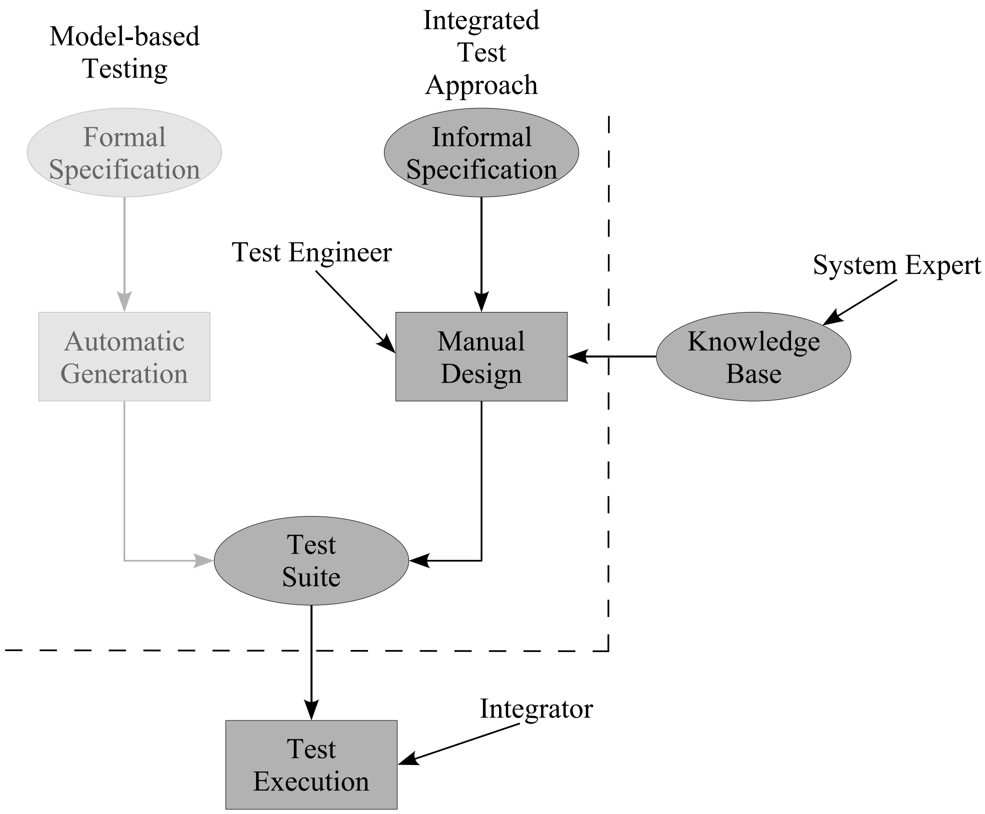


Figure 3.4: Integrated Test Approach

Whereas the trend of model-based testing and the integrated test approach is the same, i.e. automatic execution of a test suite, their starting points are quite contrary. In Figure 3.4 the two approaches are illustrated. In the first case a test suite can be generated out of a formal specification automatically. In the second case it has to be developed manually by test engineers. In both cases, however, we have to tackle the problem of test execution afterwards.

In practice it is intrinsically unrealistic to have a complete and exact formal model of a complex system in advance, as they usually comprise third-party or legacy components. Therefore "classical" model based test methods cannot be applied, since the prerequisite for such approaches, a formal specification, does not exist[1]. This is the reason why we propose an approach which supports the manual design of tests, accompanied, however, by a *Knowledge Base*, where rules concerning the construction of "correct" tests are gathered. The knowledge base reflects the expertise of senior test engineers about the do's and don'ts of test design, as well as the expertise of system experts about the correct behaviour of the considered complex system.

A key aspect of the integrated test approach is a separation of the *specification* of test cases and their actual *execution*, cf. Figure 3.4. This separation helps us to distribute the intellectual properties of "classical" test engineers among several people, i.e. to define a test process, where different experts are responsible for treating the different aspects:

— A *Test Engineer* is responsible for the design of test cases and needs therefore only knowledge concerning all aspects of the system's logic.

— An *Integrator* is responsible for the implementation of concrete test interfaces, so that test cases can be executed, i.e. covers all technical aspects.

— A *System Expert* defines frame conditions for "correct" test cases (knowledge base).

In the remainder we will first talk about the requirements for an integrated test approach (Section 3.3.1), before we define the corresponding test process (Section 3.3.2).

3.3.1 Requirements for an Integrated Test Approach

Summarizing the discussion of the previous section leads to some requirements a reasonable test approach should satisfy, so that the overall test process is supported. The resulting requirements can be classified into:

— test design,

— test verification,

— test execution,

[1]Note, however, that we present in Chapter 8 an approach to (re-)construct approximate models for special classes of complex systems.

— test run analysis, and

— general test organization

issues. In the following we discuss each category in more detail.

Test Design Requirements

As system-level testing usually treats the system under test from an end-user's point of view, this should be maintained when moving to an automated test execution, meaning that the test design should also happen at this level of expertise and intuition. In particular, it should not require programming expertise nor any knowledge of how to apply/use a specific test tool, so that it is sufficient for test engineers to be experts of the considered system's logic only. Thus an adequate test specification formalism should be on the same granularity as end-users would treat the system:

— *Flow graph* like structures which captures the essence of system runs, which are composed out of

— *coarse granular generic test building-blocks*.

The keys to the user-level test design are the test stimuli and observation points, here called *test building-blocks*, of which tests are composed. Additionally, the test building-blocks should be parameterized. Consider the example of Figure 3.1, where several telephone devices are involved in the test setting. Usually the telephones are of the same kind, so that they can perform similar actions. When providing parameterized actions (e.g. here parameterized according to the instance of the concrete telephone), less different test building-blocks are needed, and the resulting tests therefore become more intuitive.

Furthermore, the test design formalism should give test engineers enough freedom to design partially defined tests, where the overall system behaviour need not always be taken into account for test case evaluation, but it is possible to concentrate on certain aspects only.

Test Verification Requirements

Because of the complexity of the considered testing scenario, the resulting tests will become quite complex, as they must capture the global interplay between the involved components and subsystems. Open problems are e.g. the interdependencies between the involved subsystems, i.e. one test action executed on one subsystem usually results in reactions of some other. Typical examples for problems coming from these interdependencies are situations where after test execution some subsystems are in their initial state while others are not, i.e. the complete system is not in its initial state. This problem, where resources that were requested during a previous test case but not released afterwards, can influence forthcoming tests. Therefore, a knowledge base, where vital properties (*constraints*) concerning the appropriate and correct usage of parameters (local properties) as well as the interplay between the stimuli and observation points of a test (global properties) are stored, helps much to guide test engineers during test case design (cf. Figure 3.4). Design decisions that conflict with the constraints and consistency conditions of the intended system or the current test purpose can thus be detected immediately. These properties should be expressible enough to capture the essence of the expertise of senior test engineers and system experts. The constraints usually concern the do's and don'ts of test creation, e.g. which test building-blocks are incompatible, which can or cannot occur before/after some other test building-blocks, how test building-blocks are correctly be parameterized, or the correct behaviour of the considered system. Such properties are rarely straightforward, sometimes they are documented as exceptions in thick user manuals, but more often they are not documented at all, and have been discovered at a hard price as bugs in previously developed tests. The more constraints are available, the more reliable the resulting test cases are. Note, however, that it is importance in practice that the overall test approach can be applied directly, although no constraints have been defined so far (small *formalization gap*).

Test Execution Requirements

During test execution one needs to stimulate and observe all involved subsystems through several test interfaces. For this purpose usually each subsystem has its own dedicated (instance of a) test tool, cf. Figure 3.2. So distributed executed test tools of different abilities and different interconnection variants must be controlled in a way that emphasizes the aspects control of tool activities and determination of state and state changes of subsystems. The reactions of subsystems to stimuli must be retrieved and evaluated, and their evaluation results must control succeeding test steps.

Test execution should be fully automated, and, furthermore, must ensure repeatability. This is of course of intrinsic importance in the regression testing scenario. Therefore a well-defined execution semantics of test cases is needed, at least on a theoretical level, to ensure a common understanding of what test cases do. This enables test engineers to design unambiguous test cases. A concrete instance of an test environment is then committed to implement the desired execution semantics (cf. Chapter 4).

Test Run Analysis Requirements

A report must be available that records a test run and facilitates documentation and tracking of defects by providing sufficient details to support test engineers during error analysis. Note that although we are considering black-box testing it is sometimes important for a detailed analysis of the test runs to carefully document all signals of the underlying communication between the involved subsystems, which is obviously only applicable when corresponding test interfaces and test tools exist. Furthermore a characterization of the system under test, i.e., versions of subsystems and of test tools, must be documented for repetition. Result and data of each step of the tests must be logged and the overall status of a test run must be summarized.

Test Organization Requirements

Additionally an integrated test approach must also take care of several environmental aspects, i.e. the structuring and storage of tests and additional data needed throughout the test process, a generic configuration management, etc. As these requirements are not of central importance within this thesis they are discussed for the sake of completeness only.

Beside the tests themselves, many other files referenced and used in test cases have to be organized, e.g. the test building-blocks, configuration files, test documentation, or test reports. All these data evolve throughout the test process and need to be organized throughout the test process. Therefore, it is important to capture the history of changes and the dependencies between versions by means of a version control system.

Furthermore, for complex systems it is essential to prepare a test, i.e. to bring the system under test into a well-defined, initial state before a test run is started. This is because the initialization of one component usually affects the state of the others. Therefore, configuration management is needed to help during the test setup. The configuration of the test scenario ranges from the physical setup of the test infrastructure, to the concrete initialization of the system before real tests can start. Whereas the initialization of the system under test can be seen as a special test run, the physical setup cannot be automated. So configuration management consists of the administration of documentation, different versions of configuration files or even program versions resp. operating systems, and the workflow of setting up the system under test, either automated or simply documented.

Finally tests must be organized by means of several criteria, often orthogonal to each other, which particularly help managing large sets of tests. Examples for such criteria can be the test aim or the considered test scenario. For this purpose it is useful to provide a classification scheme of tests, which allows the mapping of tests according to several criteria.

3.3.2 An Integrated Test Approach – The Test Process

During the overall test process of a complex system, i.e. from the setup of the test scenario to the real execution of tests, several tasks have to be carried out. Some of them are independent of each other and can therefore be done concurrently, while dependencies exist between others, so that they have to be coordinated. In addition, in larger test projects several people with different capabilities are involved. To ensure a successful execution of the overall test project a well-defined test process is needed, one which coordinates the different tasks during the test of a complex system. Furthermore, the test process should be facilitated by an appropriate *test environment*, which will be presented in Chapter 4.

Within this section we discuss the process from the conceptual point of view, while in Chapter 5 we present the concrete tasks that have to be done from a practical point of view.

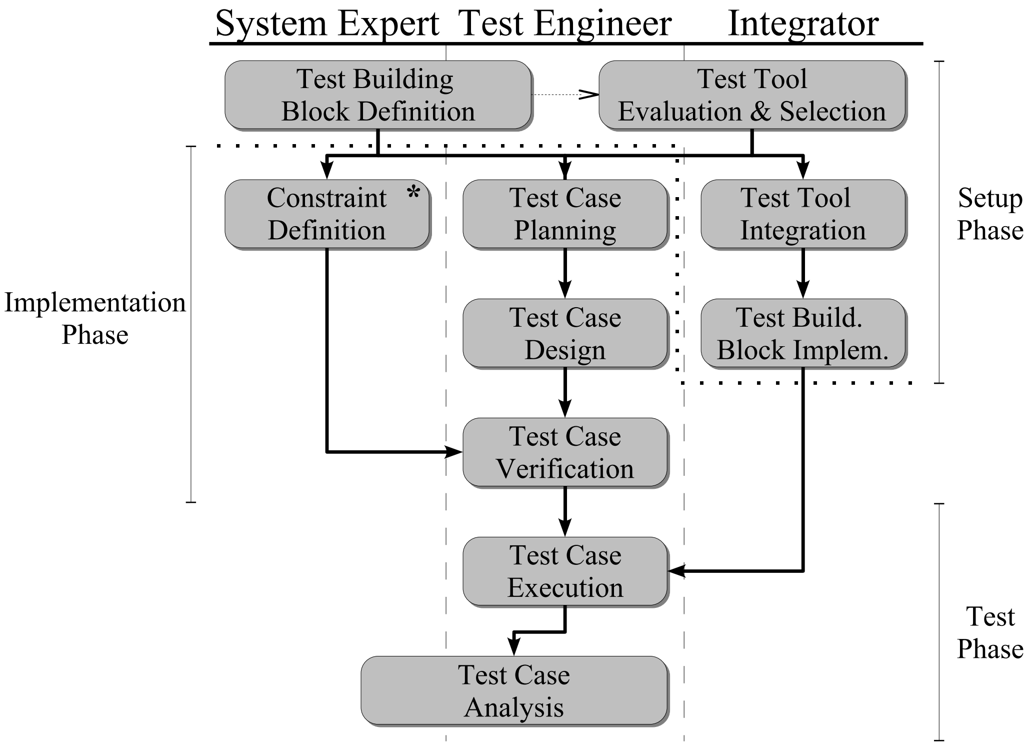


Figure 3.5: Global view of the supported test process

Figure 3.5 presents an overview of the overall test process for testing complex systems. Three different roles are involved in the process: *System Experts*, *Test Engineers*, and *Integrators* (cf. Figure 3.4). There are, however, several other participants in the test process, like *Verification Experts* or *End Users*, with minor importance.

System Expert The system expert has detailed information about the system under test; usually he is also involved in the development of the considered system. He has at least particularized knowledge of the external interfaces of the system, e.g. the graphical user interface of an application or the supported communication protocols, and the system logic itself, i.e. he can answer questions like "*how should the system react to a certain stimulus*".

Test Engineer The test engineer plays the central role within the process, as he is responsible for designing and verifying new tests, executing them, and finally analyzing the results. He also needs basic knowledge of the system under test, but less detailed than a system expert, as he will usually treat the system from an end-user point of view only.

Integrator The integrator is responsible for all environmental aspects of the process. So he has to take care of the (selection and) integration of the involved test tools and the concrete implementation of the test building-blocks, so that test cases can be executed. The integrator needs sophisticated programming skills and detailed knowledge of both the used test tools and the system under test, because he has to establish the test interfaces to the system. Since this role is so complex, it is usually shared among several team members.

Orthogonal to the involved roles we can split the process into three, usually overlapping, phases: the *setup phase*, the *implementation phase*, and the *test phase*. While in the setup phase the definition and establishment of the test interfaces take place, the implementation phase takes care of the concrete test case design. Finally, once a full test setting is integrated and the corresponding test cases are designed, the recurrent test phase starts.

In the following section we will discuss each phase in detail.

Setup Phase

The very first task within the test process is the evaluation and selection of appropriate test tools – possibly with the help of test engineers –, that can be used for the test of the system under test, by the integrator. The definition of the test building-blocks by system experts used to design test cases depends particularly on the abilities of the used test tools, as different test tools support different test actions resp. different checking capabilities. After the test tool selection, the integration into the test environment takes place, so that they can be accessed from the test execution environment. Once the test interfaces are established, the integrator can start with the implementation of the concrete test building-blocks. Here special code has to be implemented so that the test environment is able to coordinate the corresponding test tools for test execution.

Implementation Phase

Once the test building-blocks are defined, the system expert can start to define the constraints used to ensure reliable test case design. The constraints reflect typical properties of the considered scenario. To support the system experts during the specification of the constraints, we propose a pattern system in Section 3.4.4.

At the same time test engineers can start preparing the real testing of the system. Therefore, after a test planning, where the test purposes are determined, the test case design starts, which is is constantly accompanied by test case verification.

Test Phase

During the test phase the previously defined test cases can be executed automatically within the test environment. During test execution a detailed report about the test results is prepared, which is then subject to further analysis. Note that it is this area of the test process that offers the largest return of investment.

Test Engineer's View

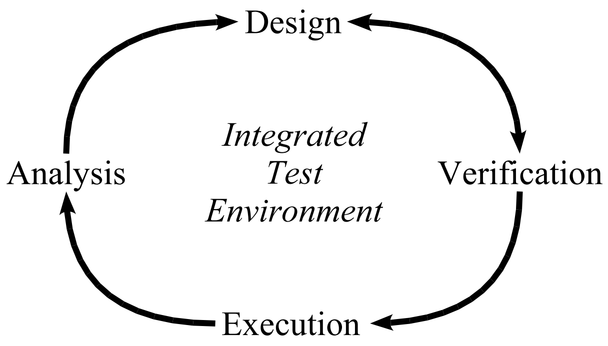


Figure 3.6: Test engineer's view of the supported test process

The test process discussed above is, however, too idealistic for practical use. For instance, a typical problem that arises during the test case design is, that some test building-blocks discovered to be missing, or the test case analysis leads to an adaption of the test case itself. So the different phases have to be organized in an iterative process. In the this section we investigate the iterative test process from the test engineer's point of view only. Therefore we assume that the setup phase and parts of the implementation phase have already been completed[2].

The iterative test process for a test engineer during the test phase is roughly presented in Figure 3.6. The cycle starts with the design of test cases. In the design phase the test engineer defines test cases on a coarse granular level, i.e. test cases that treats the system from the end-user point of view. Note that all (technical) aspects concerning, e.g. the treatment of the underlying communication protocols, have already been covered during the setup phase.

Test engineers can use test building-blocks for a graphical design of complex test cases. After the design of test cases, during the verification phase they will be checked against the constraints. The verification process provide concrete information concerning mistakes and their possible location and guide the test engineer through the redefinition of the test cases.

Test cases that pass the verification phase can be executed directly within the test environment in a control flow driven way. For this purpose executable code has been implemented for each test building-block during the setup phase, which enables the test environment to steer and observe the corresponding subsystems by the (dedicated instance of their) assigned test tool.

When executing a test case, a detailed report will be prepared in the form of a test protocol. For each test building-block executed by the execution engine, all relevant data (its execution time, its name, the version of the files associated with the test building-block, the building-block's parameter assignments, and the processed data) will be recorded in a protocol for later analysis and reuse.

The results of the analysis usually lead to a revision of the test cases themselves. This is either because incomplete or faulty test cases are detected or because adaptations to the test cases are necessary as a result of modifications in the system under test.

[2] Note that usually also tasks of the setup phase are subject of revisions, e.g. when additional test building-blocks are needed.

3.4 Formal Foundation of the Integrated Test Approach

Key aspects of the integrated test approach, presented in the previous section, are the separation of design issues like test specification and technical issues like test execution, and the support of test engineers throughout the test specification by a knowledge base. This way we are able to shift the test design issues from total experts of the system and the used test tools to experts of the system's logic. What is needed is a *flexible test specification* formalism that gives test engineers enough freedom during test case design. It is particularly important that:

— tests can be specified from an *end-user perspective*, i.e. on a level like end-users would treat the system, and

— tests can focus on *relevant system behaviour* only, i.e. take into account only special aspects of the system state, as it is intrinsically unrealistic to check the complete (distributed) state of a complex system after stimulations.

When testing complex systems we see that the formalism of tester processes, as stated in Definition 2.10, is often too finely granular. This is because tester processes have to be specified exactly to prevent from deadlock situations, as they need to be input-enabled. Additionally the test actions, of which tester processes are composed of, are on a technical level on which the tester communicates with the system.

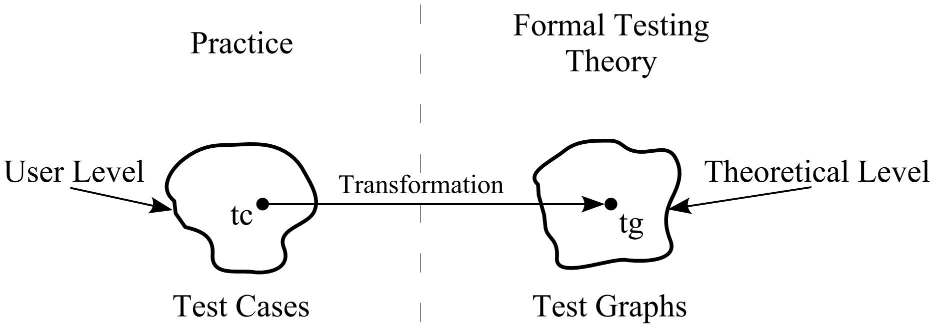


Figure 3.7: Adequate Test Case Specification

In Figure 3.7 our approach is depicted: instead of specifying tests on the fine granular level of tester processes, we propose to establish a coarse granular level for the test case design. This allows test engineers to specify test cases on the right level of adequacy, i.e. on a user lever rather than on a theoretical level. We offer, however, a transformation of test cases (user level) into (enhanced) tester processes (theoretical level), called *test graphs*, which allows us to take the well established formal testing theory, presented in Chapter 2, as a fundament for our integrated test approach. The "normal" test engineer, however, is not bothered with the underlying theory.

In the remainder we discuss in Section 3.4.1 the formalism of test cases, that provides the required flexibility. The test specification, however, preserves enough exactness to allow the definition of a precise execution semantics, which will be presented afterwards (Section 3.4.2). In Section 3.4.3 we introduce a suitable temporal logic for the specification of consistency rules (constraints), and finally we discuss how to facilitate test engineers during the formulation of their constraints by a pattern based approach (Section 3.4.4).

3.4.1 Establishment of a suitable Test Specification Formalism

We present below a notation of test cases that, beside others, provides exactly the features discussed above: introduction of coarse-granular, parameterized actions and specification of incomplete tests.

It can be observed that often the test of the interaction of several components of the same kind is the matter of interest, e.g. as can be seen in CTI systems where we treat several telephone devices with similar capabilities. Each component has the same set of commands, which can be parameterized by e.g. the concrete instance of the component, or the communicating counterpart. Here parametric actions help a great deal, as instead of providing different atomic actions for each possible variant, one parametric action is sufficient. In our context we will denote the parametric actions with *test building-blocks*, or *test blocks* for short, and test cases will be composed out of it. Test blocks are characterized by means of an identifier and a set of formal parameters. Here the parameters are bound to the test block identifier, meaning that different test block instances, but with the same identifier, have the same parameters. To simplify the parameter handling, we restrict in the following definition the domain of parameter values to integers and strings.

Definition 3.1. Let $\mathcal{N}$ be a finite fixed set of test block identifiers, and Π a finite fixed set of parameter identifiers. Let furthermore $\{Val_k\}_{k\in\mathcal{I}}$ denote the domains of parameter values, with $\mathcal{I} = \{\text{Integer, String}\}$. Then Val_{Integer} will be the set $\mathbb{Z}$ of Integers and Val_{String} the set of Strings. Val denotes the union $\bigcup_{k\in\mathcal{I}} Val_k$ of parameter value domains. The function $\texttt{type} : \mathcal{N} \times \Pi \rightarrow \mathcal{I}$ defines the types of parameters.

Test cases are represented by graphs, where formally a test case can be seen as a *kripke-transition system*, i.e. a graph where both states and transitions are labelled.

Test cases are composed out of test blocks, where we need to distinguish between *action test blocks*, that stimulates the system, and *check test blocks*, that validates possible outputs of the system. So the set of test blocks is partitioned into four disjoined sets:

1. *internal action test blocks*, which are used e.g. for test case or system under test initialization, or more general for environmental purposes;
2. *external action test block*, which stimulate the system under test;
3. *internal check test blocks*, which allow e.g. repetitive executions of (parts of) a test case with respect to internal counters; and
4. *external check test blocks*, which check the status of certain aspects of the system under test.

For a concrete instance of a test block in a test case a corresponding identifier out of $\mathcal{N}$ and a concrete parameterization of its parameters, according to their domains, has to constituted. Note that the definitions of $\mathcal{N}$, Π, and `type` are available globally, i.e. not bound to a single test case only.

Due to the intuitive semantics of the different test block classes, we can determine the number of their outgoing transitions in advance, i.e.:

— Action test blocks have one outgoing transition, and

— check test blocks have exactly two outgoing transitions (*true* if the check is ok, or *false* otherwise).

Now test cases[3] can now be defined as follows:

Definition 3.2 (Test Case). A *test case* $tc \in \mathcal{TC}$ is defined as a tuple $(\Sigma, A, \longrightarrow, s_o, \mathcal{L}, F)$, where

1. Σ is the set of available test blocks, partitioned into four pairwise disjoined sets: internal (external) action test blocks are denoted with Σ_A^I (Σ_A^E), and internal (external) check test blocks with Σ_C^I (Σ_C^E), where $\Sigma = \Sigma_A^I \cup \Sigma_A^E \cup \Sigma_C^I \cup \Sigma_C^E$. Let furthermore Σ^I denote all internal and Σ^E all external test blocks. There exists two special test blocks $\{\texttt{Passed}, \texttt{Failed}\} \subseteq \Sigma_A^I$

[3]Note that the definition of a test case refines the definition of test models used in [NSM+01] with a partition of the set of test blocks and a parameterization of test blocks. Furthermore, the set of possible labels for the transitions will be limited.

2. The set of actions is given by $A = \{default, true, false\}$

3. $\longrightarrow: \big((\Sigma_A^I \cup \Sigma_A^E) \times \{default\} \to \Sigma\big) \;\cup\; \big((\Sigma_C^I \cup \Sigma_C^E) \times \{true, false\} \to \Sigma\big)$ is a transition function[4]

4. $s_0 \in \Sigma^I$ is the uniquely determined initial test block

5. $\mathcal{L} : \Sigma \to (\mathcal{N} \times (\Pi \to Val))$ is a test block labelling function that associates a name out of $\mathcal{N}$ and a partial parameter function $p : \Pi \to Val$ with each test block

6. $F \subseteq \Sigma$ is a non-empty set of final test blocks

For technical reasons we restrict the class of all test cases to those that have reachable final test blocks, i.e. ones where at least one path from the start state to a final test block exists.

For the parameterization of test blocks we assume that all parameter values are of correct type, i.e. $\forall s \in \Sigma, \pi \in \Pi . \mathcal{L}(s) \downarrow_2 (\pi) \in Val_{\mathtt{type}(\mathcal{L}(s)\downarrow_1,\pi)} \cup \{\bot\}$, meaning that a parameter value is either undefined or of the corresponding (correct) domain. Note that $\downarrow_1$ ($\downarrow_2$) denotes the projection on the first (second) component. The parameterization of test blocks is only a syntactical construction providing further convenience to users, as we can always represent parameterized test blocks by distinguishable, generic counterparts, where each generic test block stands for one possible valuation of Π. The introduction of parameterized test blocks instead of propositional ones prevents the usage of an impractically high number of different test blocks.

Note that test cases as defined in Definition 3.2 can be seen as test programs rather than linear test sequences or tester processes. This is because during the test case design, one is not restricted introducing cycles which can be used, e.g. in conjunction with internal action and check blocks, to implement repetitions. The drawback of this flexible specification formalism is of course, that one cannot prevent users from designing infinite, and therefore counterproductive test cases, or to be more precise test cases with an infinite behaviour. We will present below a well-defined test execution semantic in terms of (special sorts of) tester processes, that is able, together with a concrete implementation of an execution engine, to handle even infinite or partially defined, test cases.

[4]Note that the transition function is defined by the union of its subfunctions. This is, however, possible because the parameter domains of the subfunctions are disjoined (*overloading*).

3.4.2 Execution Semantics for Test Cases

It is of intrinsic importance to support a well-defined execution semantics for test cases. In addition the following reasons are of particular interest:

1. Common understanding of what test cases do;
2. The handling of partially defined test cases, i.e. test cases that
   (i) do not take the complete system behaviour into account for test evaluation, or
   (ii) do not evaluate a test case at all;
3. The introduction of the notation of timeouts.

Obviously a common understanding about test cases is necessary. This is important for test engineers, who must be able to design unambiguous test cases, as well as for the implementation of the actual test environment. It becomes particularly relevant when providing a flexible test case formalism as stated in Definition 3.2, because it is not prescribed to define complete test cases only. When executing incomplete, or partial test cases, the problem occurs that during a test run additional system behaviour reflecting unexpected outputs of the system cannot be matched from within the test case and therefore the test execution deadlocks. To prevent this problem, every test case has to be completed so that for each test case a unique corresponding enhanced tester process in terms of a *test graph* has to be computed. This represents the semantics of a test cases. The test graph must also end with *pass* or *fail*, so that test runs can be properly evaluated. This must be possible even if in the test case no path from the start state to `Passed` or `Failed` exists (as it is not prescribed that such paths ever exist in Definition 3.2). Finally, a notation of timeout must be established because systems often don't respond directly to stimuli, but with a delay. When considering e.g. telecommunication systems, these timeouts are defined within the protocol specification.

In Figure 3.8 the general picture of the execution semantics is presented: First test cases will be transformed into test graphs. For test graphs we can define a notation of *test runs* similar as we have done for tester processes in Definition 2.11, i.e. we define how test graphs can be executed synchronous with a (theoretical) model of the implementation. This results in a test evaluation out of $\{pass, fail\}$, which determines the result of the original test case.

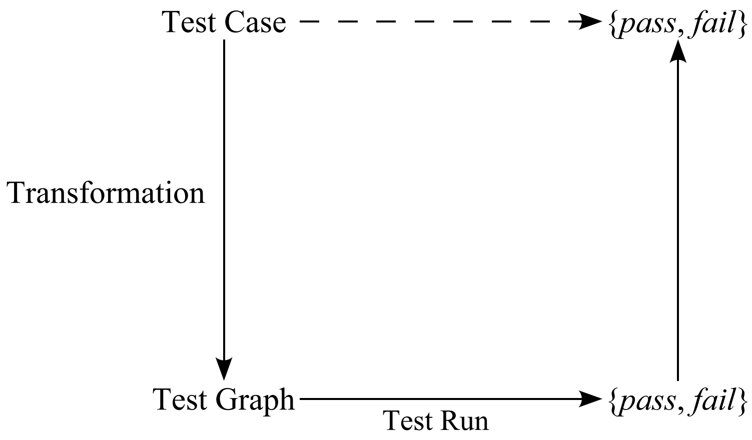


Figure 3.8: Execution Semantics of Test Cases in Terms of Test Graphs

Test Graphs and Test Runs

First we have to refine the notation of tester processes as presented in Definition 2.10, here called *test graph*. The following definition stands for a test graph with respect to its specification resp. implementation model, in terms of an input/output automata ($\mathcal{IOA}$, cf. Definition 2.7).

Definition 3.3 (Test Graph). A *test graph* $t = (\Sigma, A, \rightarrow, s_0, P)$ for an implementation $I = (\Sigma^I, A^I, \rightarrow^I, s_0^I, P^I) \in \mathcal{IOA}$ is an $\mathcal{IOA}$ ($\mathcal{TG} \subset \mathcal{IOA}$), where the following holds:

1. $A_I = A_O^I$, $A_{Int} = \{\tau, \delta\}$, and $A_O \subseteq A_I^I$
2. The special action $\delta \in A_{Int}$ denotes a timeout
3. The special states **pass** and **fail** denote terminal states, i.e. $init(\mathbf{pass}) = init(\mathbf{fail}) = \emptyset$
4. $\forall s \in \Sigma . \exists \sigma \in A^* . \left(s \xrightarrow{\sigma} \mathbf{pass}\right) \vee \left(s \xrightarrow{\sigma} \mathbf{fail}\right)$

The alphabet A of a test graph and its partition P ($A = A_I \cup A_O \cup A_{Int}$ where $A_I \cap A_O \cap A_{Int} = \emptyset$) is given by means of the alphabet of the corresponding implementation, just the other way round: the inputs of a test graph are the outputs of the specification, and the outputs are a subset of the inputs. This is particularly relevant for the compatibility of a test graph and its corresponding system. Furthermore two special internal actions enrich the alphabet to model timeouts (δ) and internal choices resp. actions of a test case (τ), which are not reflected in the communication between the test graph and the system under test. Two special states are needed for the test evaluation (**pass** and **fail**). To be exact these states are terminal states, i.e. they are not input-enabled. The condition that there exists a path from every state to either **pass** or **fail**, finally ensures that every test run can be properly evaluated.

Formally we have to adapt the definition of test runs (Definition 2.11) because we are interested in finite computations only.

Definition 3.4 (Test Run). A *test run* of a test graph $t \in \mathcal{TG}$ with an implementation $I \in \mathcal{IOA}$ is a **finite** computation of the synchronous product $t \parallel I$ leading to a terminal state of t:

$$\sigma \text{ is a test run of } t \text{ and } I =_{def} \exists I' : \left(t \parallel I \stackrel{\sigma}{\Longrightarrow} \textbf{pass} \parallel I'\right) \text{ or } \left(t \parallel I \stackrel{\sigma}{\Longrightarrow} \textbf{fail} \parallel I'\right)$$

Because of the way the alphabet of a test graph is defined and because test graphs are $\mathcal{IOA}$ as well as the specification, it is ensured that every test run terminates properly. This is because every output of the system can be matched by the test graph and vice versa, as both systems are input-enabled and because it is always possible to reach a terminal state of the test graph, cf. Definition 3.3.4.

There are two major differences between test graphs and (classical) tester processes:

— Test graphs allow *non-determinism* by means of internal actions, and

— test graphs can describe *infinite behaviour*, i.e. can contain cycles as well.

These two properties are needed to provide a flexible and user-friendly test case design. Internal actions are needed mainly for environmental purposes and a concrete test environment has to ensure a deterministic execution of such. Finite execution can be achieved by the introduction of δ, which ensures that a concrete execution engine will abort the test run after a specified period of time. Note that this common timeout captures on the theoretical level all the different, subsystem or command-specific timeouts of a concrete implementation (cf. Section 4.1).

Transformation of Test Cases into Test Graphs

The general idea behind the transformation rules of test cases into test graphs is to:

— Keep the (generic) structure of the test case graph, and

— transform test blocks into actions of the test graph.

During the transformation external action test blocks are simply translated into output actions of the test graph (i.e. input actions of the system under test), whereas check test blocks are translated into binary decisions structures, modelled by a state which is followed by two outgoing transitions. One transition is labelled with the input action, that should be produced by the system under test and is checked by the check test block, the other one is labelled with the special (internal) action δ, to model a timeout. The semantic is that we are waiting, until the timeout occurs, for the arrival of the prescribed output action and can continue accordingly. Internal test blocks are translated into internal actions of the test graph (τ), i.e. are ignored during test runs. To ensure input-enabled test graphs, reflexive transitions, labelled with all unregarded input actions, are attached to all states of the test graph. Finally, to allow a proper evaluation of test graphs, we treat every test graph as *pass* until we find a path in the test case which goes across a `Failed` test block. This information is maintained during the transformation with a temporarily valuation function ν. Note that this allow even to treat test cases, which contains no `Passed` resp. `Failed` test block.

The concrete transformation rules for test cases into test graphs are given below:

Definition 3.5 (Transformation Rules). Let $t = (\Sigma^t, A^t, \rightarrow^t, s^t_o, P^t)$ be a test graph and $tc = (\Sigma, A, \longrightarrow, s_o, \mathcal{L}, F)$ a test case. Let furthermore ν be a (temporary) valuation function which maps each state of t into a valuation, i.e. $\nu : \Sigma^t \rightarrow \{\perp, \top\}$, and $\nu(s^t_0) = \top$. $S \in \Sigma$, $A \in \Sigma^E_A \cup \Sigma^I_A$, $C \in \Sigma^E_C \cup \Sigma^I_C$, $a \in A$, $S_t \in \Sigma^t$, and $f_A : \Sigma \times \mathcal{L} \rightarrow A^t, f_\Sigma : \Sigma \rightarrow \Sigma^t$. The transformation rules of a test case into a test graph is then defined as follows:

1. $\dfrac{f_\Sigma(A) \xrightarrow{f_A(A,\mathcal{L})} f_\Sigma(S), f_\Sigma(A) \xrightarrow{A_I} f_\Sigma(A), \nu(f_\Sigma(S)) = \nu(f_\Sigma(A))}{\boxed{\text{A}} \xrightarrow{default} \boxed{\text{S}}}$

2. $\dfrac{f_\Sigma(C) \xrightarrow{f_A(C,\mathcal{L})} f_\Sigma(S), f_\Sigma(C) \xrightarrow{A_I \setminus \{f_A(C,\mathcal{L})\}} f_\Sigma(C), \nu(f_\Sigma(S)) = \nu(f_\Sigma(C))}{\boxed{\text{C}} \xrightarrow{true} \boxed{\text{S}}}$

3. $\dfrac{f_\Sigma(C) \xrightarrow{\delta} f_\Sigma(S), \nu(f_\Sigma(S)) = \nu(f_\Sigma(C))}{\boxed{\text{C}} \xrightarrow{false} \boxed{\text{S}}}$

4. $\dfrac{\nu(f_\Sigma(Passed)) = \top}{\boxed{\text{Passed}}}$

5. $\dfrac{\nu(f_\Sigma(Failed)) = \perp}{\boxed{\text{Failed}}}$

6. $$\frac{f_\Sigma(S) \xrightarrow{f_A(S,\mathcal{L})} S_t, S_t \xrightarrow{\delta} \textbf{passed}, f_\Sigma(S) \xrightarrow{A_I} f_\Sigma(S), S_t \xrightarrow{A_I} S_t}{\boxed{\mathrm{S}}} \quad \begin{array}{c} S \in F, \\ \nu(f_\Sigma(S)) = \top \end{array}$$

7. $$\frac{f_\Sigma(S) \xrightarrow{f_A(S,\mathcal{L})} S_t, S_t \xrightarrow{\delta} \textbf{failed}, f_\Sigma(S) \xrightarrow{A_I} f_\Sigma(S), S_t \xrightarrow{A_I} S_t}{\boxed{\mathrm{S}}} \quad \begin{array}{c} S \in F, \\ \nu(f_\Sigma(S)) = \perp \end{array}$$

Let us now discuss in detail what is stated in Definition 3.5.

To transform test cases into test graphs two functions are needed:

— f_A maps test blocks under consideration of their parameterization to actions of the test graph, and

— f_Σ maps test blocks to states of the test graph.

The function f_A usually maps test blocks of the same kind (i.e. with the same name) but with different parameter values (i.e. let $n_1, n_2 \in \Sigma$, then $\exists \pi \in \Pi.\mathcal{L}(n_1)(\pi) \neq \mathcal{L}(n_2)(\pi)$) into different actions[5]. An exception will be internal action and check test blocks, which will be mapped, regardless to their actual parameterization, to the internal action τ. Furthermore we often want to define f_A so that it maps one test block to a sequence of actions to establish a coarser granular view of the test setting. Without loss of generality we will assume that this kind of abstraction will take place in the definition of test cases only, i.e. several (connected) test blocks will be subsumed into one macro, which is then again available as a generic test block, cf. Section 4.4.1. Note that this point of view is for the moment only needed for technical reasons to keep Definition 3.5 as simple as possible. Theoretically it does not matter whether to use either a hierarchy on test blocks or to map single test blocks to sequences of actions.
To retain the information about cycles, the usage of the function f_Σ is required. It maps each test block, except for the two special test blocks `Passed` and `Failed`, to a corresponding state in the test graph.

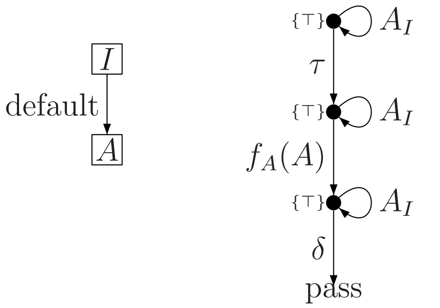
During the transformation of test cases into test graphs a valuation function ν keeps the current evaluation of the test case. Therefore, it maps every state of the test graph into $\perp$, if it is failed so far, or $\top$ if it is passed, beginning with $\top$ for the initial state. This ensures that every test run is passed until it is explicitly evaluated as failed, i.e. processes a `Failed` test block. As a consequence a test case without any evaluation will be evaluated to *pass*. The function ν is needed, because it is not prescribed for a test case where the test blocks `Passed` and `Failed` occur. So it is possible that they either occur somewhere in the middle of the test case, and

[5]Note that because $\mathcal{L}$ is available globally, for simplification we overload $f_A : \Sigma \to A^t$.

afterwards some common reset actions take place, or that they do not occur at all. In the first setting the information about the test evaluation must be propagated until to the terminal states of the test graph are reached, in the second one it must be ensured that the test graph can be evaluated at all. Note that rule 3.5.4 and 3.5.5 takes care of the update of ν.

Whereas rule 3.5.1, 3.5.2, and 3.5.3 deal with the transformation of actions resp. check test blocks, special attention is denoted to rule 3.5.6 and 3.5.7. Here for the final states of a test case with respect to ν the corresponding terminal states of the test graph will be established, i.e. *pass* resp. *fail*, where an intermediate state S_t will be introduced. Note that these two rule ensures that the transformation process terminates properly.

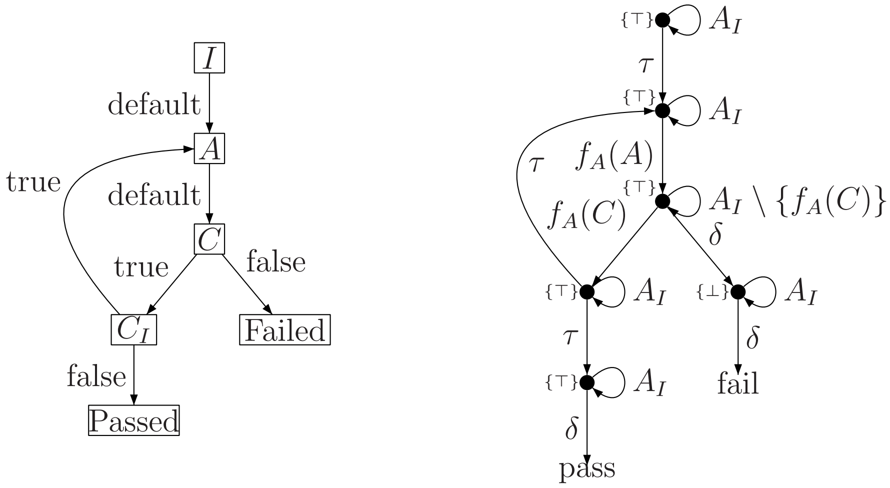
We will illustrate below the transformation using two concrete examples:



(a) Test Case (b) Test Graph

Figure 3.9: Semantic for a simple test case

Example 3.1. In the following example we present the execution semantics for the simplest test case: After the initialization (I) the system under test is stimulated by test block A (cf. Figure 3.9 (left)). When computing the corresponding test graph, we start with rule 3.5.1, followed by rule 3.5.6. Note that the initialization for the valuation function ν with $\top$ ensures that all resulting test runs will be evaluated to *pass*. In the resulting test graph (cf. Figure 3.9 (right)), one can see that at every state of the test graph, except for *pass*, a reflexive edge, labelled with all possible inputs of the tests graph, is attached to guarantee input-enableness. During test execution this ensures that all produced outputs of the system under test will be collected. Internal actions will be transformed into τ edges of the test graph, i.e. will be ignored during the communication of the test graph and the system under test.



(a) Test Case (b) Test Graph

Figure 3.10: Semantic of a test case with repetitions

Example 3.2. The example presented in Figure 3.10 shows a test case with repetitions. After the initialization of the test case by I, test action A is executed. The current system state is then validated through the check test block C. Note that it is possible that the system produces additional outputs that are not covered by C. When the checked output does not occur within a predefined period of time (δ), the test case continues with *Failed*, otherwise an internal check test block repeats the execution of A with respect to an internal counter, i.e. if the counter is less than a specified value that can be defined through a test block parameter, A will be executed, or otherwise the test case ends with *Passed*. Here the test execution environment has to ensure that in the corresponding test graph the τ-edge which leads to *pass* will sometimes be chosen.

Test cases in conjunction with their execution semantics in terms of test graphs can be specified partially only. Their semantics ensures that they can be executed without deadlocking. This enables test engineers to design their test cases aspect oriented, i.e. to concentrate for the test evaluation on certain aspects of the system state only instead of taking the overall system behaviour into account, which is obviously often too finely granular when testing complex systems.

3.4.3 Verifying Test Cases

Test cases are subject to *local* and *global constraints* which together offer a means to identify "critical" pattern in the test case during the early design phase. The semantic domain of the constraints is given by the test blocks together with their actual parameterization. Within *local constraints* assertions concerning certain test block parameter values can be expressed, e.g. in the simplest case that a parameter value is not nil. Other examples for local properties can be structural properties, like checking the number of outgoing edges of a test block. Local constraints are characterized by the fact that they can be checked within the context of a single test block. There are, however, more sophisticated properties that captures the global interplay of test blocks, called *global constraints*. Using global constraints allows users to specify causality, eventuality and other relationships between test blocks which may be necessary in order to guarantee frame conditions for e.g., executability and version compatibility. Typical examples of these kinds of constraints are

— *general ordering properties*, in which
a test block must be executed/reached some time before some other test block,

— *abstract liveness properties*, in which
a certain test block is required to be executed/reached eventually, and

— *abstract safety properties*, in which
two specific test blocks must never occur within the same test run.

Note that usually not only the test blocks themselves are taken into account, but also their actual parameterization. Within global constraints essential properties of correct test runs resp. properties that capture what is forbidden in a test run can be described.

As the treatment of local constraints is rather straightforward, we will concentrate here on the more interesting global properties. Therefore, we present a formalism which is suitable for capturing global constraints.

Global constraints are specified in a temporal logic, to be exact in a variant of the linear-time temporal logic [Eme90, Sti92], called *Semantic Linear-time Temporal Logic* (*SLTL*), cf. [SM99, NSM$^+$01]. In the following we present a first-order extension of *SLTL* called *Extended Semantic Linear-time Temporal Logic* (*ESLTL*), capable of dealing with parameterized models, in the style of [Hof97], where a first-order extension of the modal μ-calculus is discussed.

For the definition of the atomic propositions that can then be used in the specification of the constraints, there exists two possibilities:

1. Each test block of a test case has a corresponding atomic proposition determined by its name, or
2. an atomic proposition take additionally the parameterizations of a test block into account.

As mentioned above, it is useful to take test blocks together with their actual parameterization into account; therefore, we will concentrate below on the second variant only. The first case, however, can also be seen as a special case of the second one.

The set $\mathcal{A}$ of atomic propositions is then as follows:

$$\mathcal{A} =_{def} \{(n, \pi, c, i) | n \in \mathcal{N}, \pi \in \Pi, c \in C, \text{and } i \in Val_{\mathtt{type}(n,\pi)}\}$$

where $C =_{def} \{=, \neq, \leq, \geq, <, >\}$.

Intuitively an atomic proposition consists of a name n for a test block and a parameter π, which is compared by means of a comparator c to a value i. So atomic propositions can be parameterized by means of the parameter value i. Therefore instead of using atomic propositions directly, we will use atomic formulae of the form $p(t)$ instead, where p is an unary predicate symbol, and t is either a value constant c_i, or a value variable x ranging over an arbitrary domain $\mathcal{I}$ of values. Let $\mathcal{P}$ be the set of unary predicate symbols, where p denotes a single one.

Now we are ready to define the syntax and semantics of the first-order extension of *SLTL*.

Definition 3.6 (Syntax of Extended Semantic Linear-time Temporal Logic).

The syntax of extended Semantic Linear-time Temporal Logic (*ESLTL*) is given in BNF format by:

$$\Phi ::= \quad p(t) \mid \neg\Phi \mid (\Phi \vee \Phi) \mid \mathbf{<} a \mathbf{>}\, \Phi \mid \mathbf{X}(\Phi) \mid \mathbf{G}(\Phi) \mid (\Phi\ \mathbf{U}\ \Phi) \mid \exists x(\Phi)$$

where a is given as a propositional logic formula over the set of actions A, and $p(t)$ is a predicate symbol out of $\mathcal{P}$, where t can be either a value constant c_i or a value variable x ranging over an arbitrary domain $\mathcal{I}$ of values.

The *ESLTL* formulae will be interpreted over the set of all legal paths of a test case which are given by sequences of test blocks. The semantics of *ESLTL* formulae is intuitively defined as follows:

— $p(t)$ is satisfied by every path whose first element (a test block) satisfies the predicate $p(t)$, meaning that both the name of the test block matches and the parameterization is as prescribed.

— Negation $\neg$ and disjunction $\vee$ are interpreted in the usual fashion.

— Next-time operator $< >$:
$< a > \Phi$ is satisfied by all paths where the action of the transition from the first to the second element satisfies a and whose continuation, i.e. starting from the second test block, satisfies Φ. In particular, $< true > \Phi$ is satisfied by every test sequence whose continuation satisfies Φ.

— Next-step operator $\mathbf{X}$:
$\mathbf{X}(\Phi)$ requires that Φ is satisfied for the next situation in the path.

— Generally operator $\mathbf{G}$:
$\mathbf{G}(\Phi)$ requires that Φ is satisfied for every suffix.

— Until operator $\mathbf{U}$:
$(\Phi \ \mathbf{U} \ \Psi)$ expresses that the property Φ applies at all test blocks of the sequence, until a position is reached where the corresponding continuation satisfies the property Ψ. Note that $\Phi \ \mathbf{U} \ \Psi$ guarantees that the property Ψ eventually holds (strong until).

— Exists operator:
$\exists x(\Phi)$ expresses that there exists a value c_i for x such that Φ holds, if every x is replaced by c_i.

In addition, the following derived operators can be defined for further convenience:

Conjunction:	$(\Phi_1 \wedge \Phi_2)$	$=_{def}$	$\neg(\neg\Phi_1 \vee \neg\Phi_2)$
Implication:	$(\Phi_1 \Rightarrow \Phi_2)$	$=_{def}$	$\neg\Phi_1 \vee \Phi_2$
Forall:	$\forall x(\Phi)$	$=_{def}$	$\neg\exists(\neg\Phi)$
Box:	$[a]\Phi$	$=_{def}$	$\neg< a > (\neg\Phi)$
Eventually:	$\mathbf{F}(\Phi)$	$=_{def}$	$\neg\mathbf{G}(\neg\Phi)$
Weak Until:	$(\Phi \ \mathbf{WU} \ \Psi)$	$=_{def}$	$(\Phi \ \mathbf{U} \ \Psi) \vee \mathbf{G}(\Phi)$

Whereas the first three operators are quite obvious, the latter ones need some explanation. The *eventually* or *finally* operator $\mathbf{F}$ ensures that Φ holds for some (later) situation. The *weak until* operator $\mathbf{WU}$ holds, in contrast to the *strong until* operator $\mathbf{U}$, even when Φ holds forever.

Before we are able to define a formal semantic for *ESLTL*, we need to state a valuation function $\mathcal{V}$, that can be used to compute a set of nodes that fulfil a certain predicate p with respect to a value i.

Definition 3.7 (Valuation Function). The *valuation function* $\mathcal{V} : \mathcal{P} \to (\mathcal{I} \to 2^{\Sigma})$ is given through:

$$\mathcal{V}((n,\pi,c))(i) =_{def} \begin{cases} \left\{ s \in \Sigma \;\middle|\; \begin{array}{l} \mathcal{L}(s) \downarrow_1 = n, \text{ and} \\ (\mathcal{L}(s) \downarrow_2 (\pi), i) \in c^{\sim} \end{array} \right\} & \text{if } \mathcal{L}(s) \downarrow_2 (\pi) \text{ is defined,} \\ \emptyset & \text{otherwise.} \end{cases}$$

where $c^{\sim} \subseteq Val \times Val$ denotes the binary relation c, i.e. $=^{\sim}$ is equality on integers (strings), $\neq^{\sim}$ is inequality of integers (strings), and so on.

$\mathcal{V}$ defines for each predicate p and value $i \in \mathcal{I}$, in which states $p(i)$ holds. Note that the set $\mathcal{A}$ of atomic propositions is infinite, as the parameter domains are infinite. To check a certain property ϕ it is sufficient to consider the finite restriction $\mathcal{V}|_{\mathcal{A}(\phi)}$ only, where $\mathcal{A}(\phi)$ denotes the set of atomic propositions appearing in ϕ.

The formal definition of the semantics, with respect to a test case, is given as follows, where π denotes a path of test blocks, and π_i the i-th position.

Definition 3.8 (Semantics of Extended Semantic Linear-time Temporal Logic).

$$\begin{array}{lll}
\pi \models p(t) & \Leftrightarrow_{def} & \begin{cases} \pi_0 \in \mathcal{V}(p)(i) & \text{if } t = c_i \\ \texttt{false} & \text{otherwise} \end{cases} \\
\pi \models \neg\Phi & \Leftrightarrow_{def} & \pi \not\models \Phi \\
\pi \models (\Phi_1 \vee \Phi_2) & \Leftrightarrow_{def} & \pi \models \Phi_1 \text{ or } \pi \models \Phi_2 \\
\pi \models \langle a \rangle \Phi & \Leftrightarrow_{def} & |\pi| > 1 \text{ and } \pi_0 \xrightarrow{a} \pi_1 \text{ and } \pi_1 \models \Phi \\
\pi \models \mathbf{X}(\Phi) & \Leftrightarrow_{def} & |\pi| > 1 \text{ and } \pi_1 \models \Phi \\
\pi \models \mathbf{G}(\Phi) & \Leftrightarrow_{def} & \text{for all } k \text{ with } 0 \le k < |\pi|, \pi_k \models \Phi \\
\pi \models (\Phi \;\mathbf{U}\; \Psi) & \Leftrightarrow_{def} & \text{there is } k,\ 0 \le k < |\pi|, \text{ with } \pi_k \models \Psi \text{ and for all } i, \\
 & & 0 \le i < k, \pi_i \models \Phi \\
\pi \models \exists x(\Phi) & \Leftrightarrow_{def} & \bigvee_{i \in \mathcal{I}} [i/x]\Phi
\end{array}$$

The substitution $[i/x]\Phi$ is used to specify the semantics of quantified sub-formulae, and is itself defined inductively. It replaces all occurrences of value variable x with the constant c_i. It ensures correct scoping, as the recursive process stops at a quantified sub-formula binding value variable x.

Definition 3.9.

$$
\begin{array}{lll}
[i/x]p(t) & =_{def} & \begin{cases} p(c_i) & \text{if } t = x \\ p(t) & \text{otherwise} \end{cases} \\
[i/x]\neg\Phi & =_{def} & \neg[i/x]\Phi \\
[i/x](\Phi_1 \vee \Phi_2) & =_{def} & [i/x]\Phi_1 \wedge [i/x]\Phi_2 \\
[i/x]\!<\!a\!>\Phi & =_{def} & <\!a\!>[i/x]\Phi \\
[i/x]\mathbf{X}(\Phi) & =_{def} & \mathbf{X}[i/x]\Phi \\
[i/x]\mathbf{G}(\Phi) & =_{def} & \mathbf{G}[i/x]\Phi \\
[i/x](\Phi\ \mathbf{U}\ \Psi) & =_{def} & [i/x]\Phi\ \ \mathbf{U}\ \ [i/x]\Psi \\
[i/x]\exists x'(\Phi) & =_{def} & \begin{cases} \exists x'[i/x]\Phi & \text{if } x \neq x' \\ \exists x'\Phi & \text{otherwise} \end{cases}
\end{array}
$$

To model check an *ESLTL*-formula with respect to a test case, both the formula and the test case will be transformed first. Within this transformation step the predicates of the formula will be transformed into (real) atomic propositions again (i.e. without parameters). Furthermore, corresponding atomic propositions have to be attached to all test blocks of the test case. Afterwards the properties can be checked within a standard, propositional model checking algorithm. Intuitively the problem of checking $\mathcal{TC} \models_{\mathcal{P}} \Phi$, where we need to check a parameterized formula with a special checking algorithm, will be transformed into $\mathcal{TC}' \models \Phi'$, where we transform both the model and the formula, but can use a standard checking algorithm. A detailed technical discussion of the transformation of both the model and the formula to use standard model checking techniques can be found in [Hof97].

3.4.4 Pattern System for Property Specification for Finite State Verification

Although the formalism of *ESLTL* is, to some extent, given by a user-friendly notation, it is not suitable to be used by "normal" test engineers. This is the reason why we propose a pattern based approach, which allows an easy specification of such properties. Within a survey, Dwyer et. al. have evaluated more than 500 examples of property specifications, and they have found out that the vast majority (92%) are instances of only a few patterns, which they organizes in a pattern system for property specification for finite state verification [DAC98, DAC99]. With this pattern system it becomes quite simple to define properties, and particularly it enables even people, unexperienced with temporal logic to formulate such. We present the original pattern system, and discuss how to extend it, so that it properly fits to our needs, i.e.:

— *Parameter Handling*, so that particularly *ESLTL* can be covered, and

— *Grouping of Patterns*, to capture more sophisticated relations between patterns.

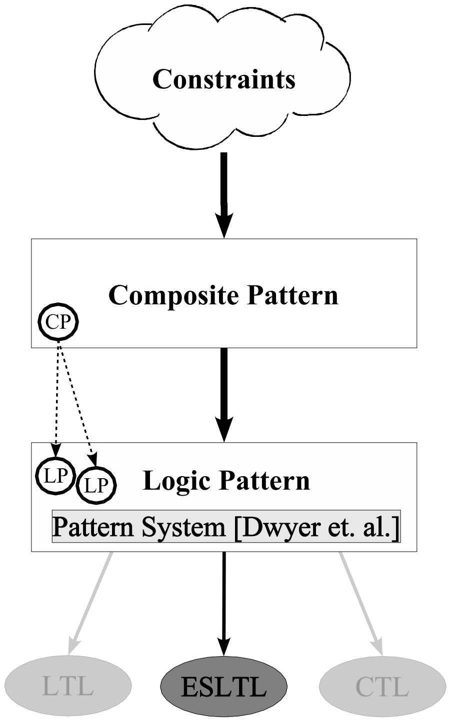


Figure 3.11: Overview of the pattern system

Figure 3.11 illustrates the general idea behind our enhanced pattern system. It bases on the original pattern system of [DAC98, DAC99], however enriched with the parameter handling (*Logic Pattern*). Note that within this thesis we are only interested in a mapping of the logic pattern to *ESLTL*, but it is not restricted to it. One can observe that often the interdependencies between certain states/events cannot usefully be captured within a single property, e.g. the following two properties belong strictly together "*After a user logs in to the system, he has to logout afterwards*" and "*A user can logout only after he has been logged in before*". Therefore we propose the notation of *composite pattern*, which are capable of capturing a more sophisticated relationship between states/events. These composite pattern are defined on top of the logic pattern and allows to group them. Technically speaking a composite pattern consists of the conjunction of several logic pattern. Finally typical users, e.g. test engineers, can instantiate composite pattern to define concrete *Constraints*, which can then be used for test case verification.

We will first present the original pattern of Dwyer et. al., before we discuss our enhancements in more detail.

Pattern System

Dwyer et. al. presented in [DAC98, DAC99, DAC97] a pattern system for property specifications for finite state verification. In their work they developed a pattern system, based on property specification patterns together with various scopes, and a corresponding hierarchy, cf. Figure 3.13. Such patterns describe the essential structure of a system's behaviour and provide expressions of this behaviour in a range of common formalisms. For each pattern and a certain scope they present mappings for several temporal logics, e.g. beyond others for linear-time temporal logic (LTL), computational tree logic (CTL), and the regular alternation-free μ-calculus. Because different specification formalisms are either event based or state based, the patterns make arrangements for both of them.

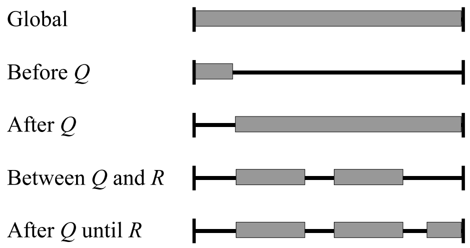


Figure 3.12: Pattern Scopes [DAC99]

Each pattern consists of a name and intent, as well as a *scope*, over which it must hold. Figure 3.12 illustrates the portions of an execution that are designated by the different scope types. Each scope is determined by specifying a starting and an ending state/event for the pattern. There are five basic kinds of scopes:

Global The property holds globally.

Before The property holds before the occurrence of a certain state/event.

After The property holds after the occurrence of a certain state/event.

Between The property holds in between the occurrence of one given state/event up to another one.

After-Until As above, but it even holds if the second state/event does not occur at all.

The property patterns are as follows:

Absence A given state/event does not occur within a scope.

Existence A given state/event must occur within a scope.

Bounded Existence A given state/event must occur k times within a scope.

Universality A given state/event occurs throughout a scope.

Precedence A state/event P must always be preceded by a state/event Q within a scope.

Response A state/event P must always be followed by a state/event Q within a scope.

Chain Precedence A sequence of states/events $P_1, \ldots, P_n$ must always be preceded by a sequence of states/events $Q_1, \ldots, Q_m$.

Chain Response A sequence of states/events $P_1, \ldots, P_n$ must always be followed by a sequence of states/events $Q_1, \ldots, Q_m$.

In Figure 3.13 a hierarchy is presented, which organizes the patterns. Here the patterns are distinguished according to whether they deal with the occurrence or the ordering of events/states.

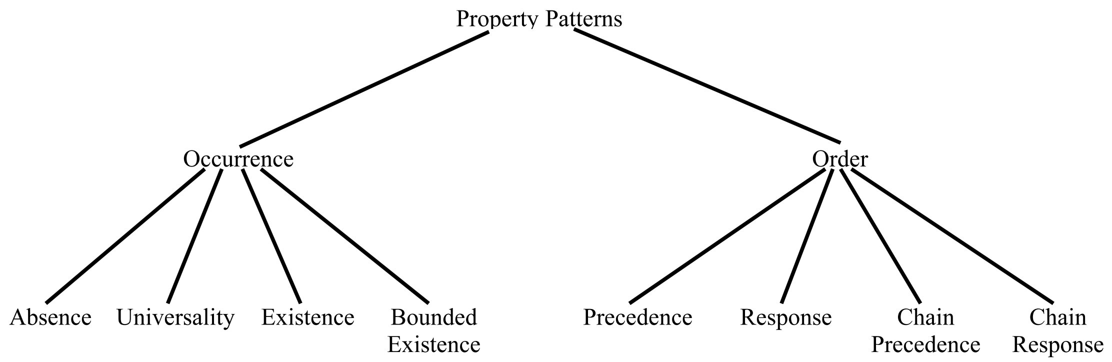


Figure 3.13: Pattern Hierarchy [DAC99]

In principle each pattern of the pattern system proposed by [DAC99] consists of five concrete ones, where there is one instance for each scope. In Figure 3.12 it can be seen that in addition to the placeholders of the patterns themselves, the scope introduces new dependencies needed e.g. to define the start and end point of the scope *between*. So each concrete pattern P depends on a number of placeholders A_1 to A_n, which will be abbreviated by $P(A_1, \ldots, A_n)$[6]. The set of all concrete pattern is called *logic pattern* and is denoted with $\mathcal{LP}$. Note that when we talk about a pattern below we will always mean a concrete pattern.

Enhancements of the Pattern System

We propose two enhancements of the original pattern system of [DAC99]. The improvements are quite orthogonal to each other and can therefore be applied independently. In particular, we present an extension of the patterns itself, so that we are able to take advantage of the first-order portions of our logic. Handling the first-order portions of *ESLTL* requires to introduce another dimension within each pattern, i.e. to allow the usage of the first-order quantifiers. On the other side we extend the overall pattern system to support the handling of groups of patterns.

Parameter Handling

The introduction of parameterized propositions instead of atomic propositions is itself rather straightforward, as the placeholders in the pattern can already deal with it. The use of the two first-order quantifiers, however, introduces another dimension within the patterns. Now a pattern consists, beyond other information, of a scope and of a quantification over parameters.

To obtain a first-order pattern out of an original one, the following transformation step has to be performed:

$$\mathcal{Q}x.(P(A_1, \ldots, A_n)[A_1/A_1^p(x_1)] \ldots [A_n/A_n^p(x_n)]), \text{ with } \mathcal{Q} = \{\exists, \forall\}$$

Intuitively within each pattern all placeholders will be replaced with parametric ones, and additionally the pattern will be embedded with a quantifier. Note that if $x \neq x_1, \ldots, x \neq x_n$ the constraint is treated like a “classical” propositional one.

Example 3.3. In the following example we will illustrate how to construct a parameterized pattern out of a propositional one within the *Response* pattern with scope *Global*. The original mapping of this pattern to *LTL* is

$$\mathbf{G}(P \Rightarrow \mathbf{F}(S))$$

[6] Note that the logical variables A_i will be substituted by single atomic propositions or simple propositional formula only.

First of all we have to substitute each occurrence of an atomic proposition with a parameterized one, i.e. P will be replaced by $P(x_p)$, and S by $S(x_s)$. After that, we will embed the whole pattern into a quantifier, which results in

$$\mathcal{Q}x(\mathbf{G}(P(x_p) \Rightarrow \mathbf{F}(S(x_s)))), \text{ where } \mathcal{Q} = \{\exists, \forall\}$$

The property that every *login* for a certain user must be followed by a *logout* can now be specified as

$$\forall x(\mathbf{G}(login(x) \Rightarrow \mathbf{F}(logout(x))))$$

i.e. we use the $\forall$-quantifier, and $x = x_p = x_s$.

The complete mappings for *ESLTL* of all patterns can be found in Appendix A.

Handling groups of patterns

It is apparent that the relationship between states/events are often manifold, and that they cannot be captured within a single logic resp. concrete pattern. Therefore it is useful to establish the notation of *Composite Pattern* which allow us to combine logic patterns. The level of *Logic Pattern* corresponds to the concrete pattern of [DAC98, DAC99], i.e. in fact, the patterns proposed are a subset of our logic patterns, because they represent the propositional case only (Figure 3.11). Logic patterns, as well as their mapping to a concrete logic, are usually defined by temporal-logic experts and can be seen as the fundament of the overall pattern system. Composite patterns are specified on top of the logic pattern and combine them, meaning that we have a $1 : n$ relation between composite patterns and logic patterns. As composite patterns already deal with predefined logic patterns, they can be defined domain specific by experts within the considered application domain, without deeper knowledge of the underlying logic.

Formally a composite pattern is a conjunction of several logic, or concrete, patterns, where a mapping of the variables of the composite pattern to variables of the logical patterns will additionally be established.

Definition 3.10 (Composite Pattern). Let $\mathcal{LP}$ denote the set of all logic pattern, and a subset $L = \{P_1(A_1^1, \dots, A_1^{r_1}), \dots, P_s(A_n^1, \dots, A_s^{r_s})\} \subseteq \mathcal{LP}$. Then P is called a *composite pattern*, where

$$P(B_1, \dots, B_t) = \bigwedge_{1 \le i \le s} P_i(A_i^1, \dots, A_i^{r_i})$$

where $A_1^1 = B_i, \dots, A_s^{r_s} = B_j$ with $1 \le i, j \le t$.

Note that in Section 4.4.2 we propose a notation to specify composite pattern.

Example 3.4. Let us illustrate the definition of composite patterns according to a simple example, where we capture the relationship between the request req and release rel of a resource. The relation between these actions is twofold, as on the one hand every requested resource should be released at some point, and on the other hand a resource can only be released, when it was previously requested. The corresponding patterns are both with scope *Global*, the *Response* pattern, i.e.

$$P^{glob}_{Resp}(P_1, S_1) = \mathbf{G}(P_1 \Rightarrow \mathbf{F}(S_1))$$

and the *Precedence* pattern

$$P^{glob}_{Prec}(P_2, S_2) = \neg P_2 \ \mathbf{WU} \ S_2$$

The notation of behavioural pattern allows the combination of the two, i.e.

$$P_B(req, rel) = P^{glob}_{Resp}(P_1, S_1) \wedge P^{glob}_{Prec}(P_2, S_2)$$

where

$$P_1 = S_2 = req \ \text{ and } \ P_2 = S_1 = rel$$

This results in

$$P_B(req, rel) = (\mathbf{G}(req \Rightarrow \mathbf{F}(rel))) \wedge (\neg rel \ \mathbf{WU} \ req)$$

Chapter 4

An Integrated Test Environment

To facilitate the integrated test approach, presented in the previous chapter, a tool support is unavoidable. In particular, a modular and open test framework is needed so that diverse test tools and units can be integrated as required. Combining test tools allows us to make optimal use of their individual strengths and to avoid their weaknesses by using the most adequate tool for each task.

Within this chapter we discuss the implementation fundamentals of an integrated test environment (*ITE*). In Section 4.1 we propose a general architecture for an *ITE* for the test of complex systems. We were able to build the *ITE* on top of an existing general purpose environment for the management of complex workflows as is presented in Section 4.2. The communication architecture of the *ITE* and particularly the notation of test tools is presented in Section 4.3. After that we discuss in Section 4.4 how the requirements presented in Section 3.3.1 are implemented in practice. Section 4.5 introduces the test tool integration task. This chapter concludes with the discussion of the integration of a special test tool (Section 4.6) capable of stimulating and observing graphical user interfaces. It is generic enough to be of use in various test settings.

4.1 Designing an Integrated Test Environment

In system-level testing the involved test tools are usually distributed and run on heterogenous platforms, cf. Figure 3.2. Therefore a flexible architecture for a test environment should lead to a modular and open environment, so that diverse test tools and units under test can be added as required. A sophisticated framework for a test environment must support a uniform handling of such test tools and provide a well-defined, preferably almost automated, integration process. This comprises the "core" communication between the test environment and the test tools, as well as the appropriation of the offered test stimuli and observation points to test engineers for test case design. To support this test tools must provide a special interface, preferably in a common used formalism like the *Interface Definition Language* (IDL). The interface then forms the basis for both the integration into the framework and for the definition of building-blocks for test case design.

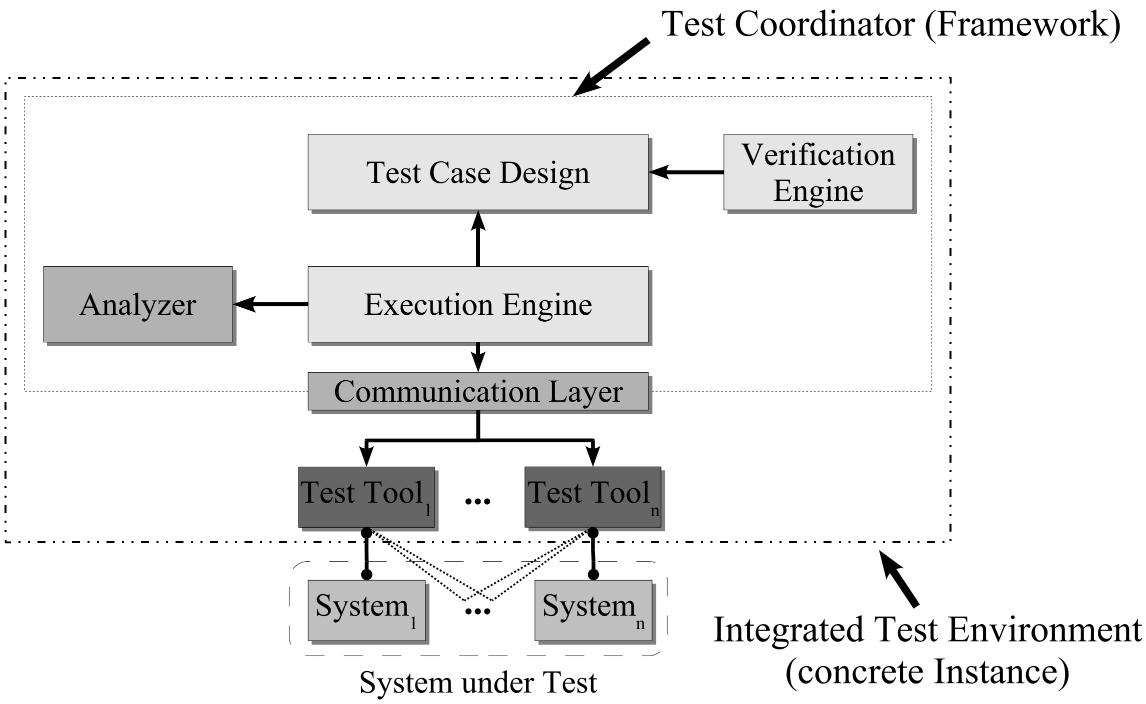


Figure 4.1: Architectural overview of the integrated test environment

The classical test execution architecture presented in Chapter 2 is not suitable for handling complex systems. This is because such systems cannot be tested within a single monolithic tester or test tool. Instead we have to coordinate several independent test tools, each suitable for testing a specific aspect, e.g. a certain subsystem, or a specific communication protocol. In general, a test tool is able to control several *points of control and observation* (PCO's) and to observe several *points of observation* (PO's) simultaneously. Altogether the different test tools, probably running on heterogenous platforms, allow the testing of the overall complex system.

Figure 4.1 presents a rough overview of an architecture for an integrated test environment (*ITE*). The system under test is composed of several subsystems, here denoted by $System_1$ to $System_n$. The subsystems are controlled and observed by the corresponding test tools. Sometimes a one-to-one mapping is possible between a subsystem and a test tool, but in general a single test tool has access to several P(C)O'S, denoted by the dashed connections in Figure 4.1. The test tools themselves are coordinated by the *ITE* more precisely by the execution engine, by means of a communication layer.

The *ITE* consists of two main portions: the general framework, also called core *ITE* or *test coordinator*, and the concrete instantiation, where test interfaces for specific complex systems are integrated.

Integrated Test Environment – The Framework

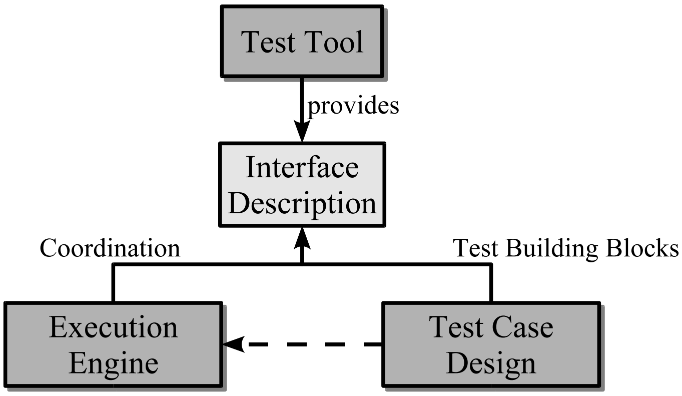


Figure 4.2: Concept of the integration of a test tool into the test environment

The core *ITE* can be seen as a framework consisting of the following modules: *Test Case Design*, *Execution Engine*, *Verification Engine*, *Analyzer*, and the *Communication Layer*. We also often refer to this as the *Test Coordinator*.

In Figure 4.1 the modules and their dependencies can be seen. The design module enables test engineers to graphically construct tests out of test building-blocks, which have to be defined separately, cf. Section 3.3.2. The verification engine checks the resulting tests with respect to a set of constraints, i.e. the verification engine has access to the designed tests. The execution engine is in principle able to execute the tests by means of a communication layer. Within the framework the execution engine only defines generic interfaces to coordinate test tools, where a concrete test tool provides its particular functionality, specified via an interface description (e.g. expressed in the *IDL*), cf. Figure 4.2. The *Interface Description* forms the basis for the integration of the test tool functionality into the test environment, meaning that the functionality can be accessed by the execution engine during test execution. Furthermore the *Interface Description*, or more precisely the abilities of the corresponding test tool, determines what test actions and observations points are available for test case design. After the integration of concrete test interfaces the framework can be used for testing concrete complex systems. In general this approach introduces the required flexibilization of the overall architecture of the test environment. When integrating a new system under test, the required test tools can be easily added. Via the test tools the *ITE* has access to all the involved subsystems and can manage the test execution by a coordination of different, heterogenous test tools.

Finally, during test execution an analyzer module records all relevant data that is needed for the detailed analysis of the test runs afterwards. This can range from the information which test building-blocks have been executed in which order, i.e. the concrete test run, up to a detailed record of the signals of the involved communication protocols.

Concrete Instance of the Integrated Test Environment

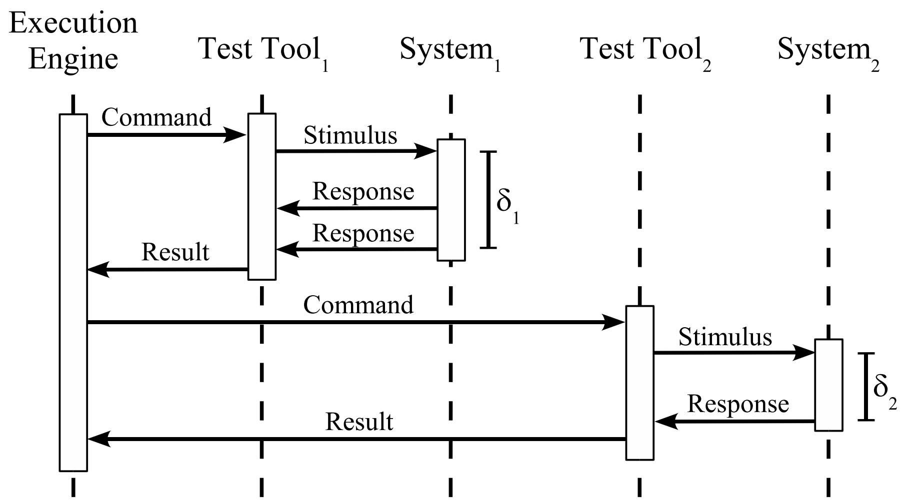


Figure 4.3: Coordination of involved test tools

When testing a specific complex system, it was previously explained that concrete test interfaces to the subsystems have to be established for the *ITE*. So a concrete instance of the *ITE* consists, beside the core framework, of several test tools. These test tools will be coordinated during test execution by the execution engine of the *ITE*. Figure 4.3 demonstrates how the coordination task takes place: The execution engine triggers $Test\ Tool_1$ with a certain command. The test tool itself stimulates the dedicated subsystem(s) (here $System_1$) and afterwards collects all the responses. To ensure that all relevant responses will be picked up, the test tool waits for a subsystem or even command-specific timeout δ_1. The collected data will be preprocessed, and sometimes an abstraction takes place. Afterwards the result will be sent back to the execution engine. The analysis of the result determines which action/observation should next take place. Here the next command stimulates $System_2$

through *Test Tool*$_2$, in the same manner as before. However, the timeout for the second command is in general different to the previous one (δ_2). This synchronous execution of test actions – i.e. stimulation of the system under test and waiting a certain period of time to ensure that the system has produced all its responses – is needed to ensure repeatable test results.

4.2 The Agent Building Center

The *ITE* or more precisely the test coordinator, is built on top of an existing general purpose environment for the management of complex workflows, (METAFrame Technologies' *Agent Building Center* (*ABC*)) [SM99], which already encompass some of the requirements that are discussed in Section 3.3.1 in an application independent way. The *ABC* is made up of the following characteristics:

Behaviour-Oriented Development: In general, application development in the *ABC* consists of a behaviour-oriented combination of building-blocks on a *coarse* granular level. Building blocks are identified on a functional basis, understandable to application experts, and usually encompass a number of "classical" programming units (be they procedures, classes, modules, or functions). They are organized in application-specific collections (palettes). In contrast to (other) component-based approaches, e.g., for object-oriented program development, *ABC* focusses the dynamic behaviour: (complex) functionalities are graphically stuck together to yield flow graph-like structures embodying the application behaviour in terms of control. This graph structure is independent of the paradigm of the underlying programming language. In particular, we view this flow-graph structure as a control-oriented coordination layer on top of data-oriented communication mechanisms enforced, e.g., via RMI, CORBA or (D)COM. Concretely, the test coordination layer communicates with individual test tools by means of CORBA [Obj99]. Accordingly, the purely graphical combination of building-blocks' behaviours happens at a more abstract level.

Incremental Formalization: The successive enrichment of the application-specific development environment is two-dimensional. In addition to the library of application-specific building-blocks, a collection which dynamically grows whenever new functionalities are made available, *ABC* supports the dynamic growth of a hierarchically organized library of *constraints*, controlling and governing the adequate use of these building-blocks within application programs. This library is intended to grow with the experience gained while using the environment, e.g., detected errors, strengthened policies, and new building-blocks may directly lead to additional constraints.

It is the possible *looseness* of these constraints which makes the constraints highly reusable and intuitively understandable.

Library-Based Consistency Checking: Throughout the behaviour-oriented development process, *ABC* offers access to mechanisms for the verification of libraries of constraints via model checking. The model checker individually checks hundreds of usually very small and application- and purpose-specific constraints over the flow graph structure. This allows concise and comprehensible diagnostic information in the case of a constraint violation, particularly as the information is given at the application rather than at the programming level.

The *ABC* is a well established toolkit, successfully applied both in academia and industrial practice. It forms particularly the kernel for the *Electronic Tool Integration Platform* [CSMB97, SMB97, Bra01]. Furthermore, it is used for the design of *Intelligent Network Services* [SMC+96a, SMC+96b, SMBK97, BMSY97, SMCB96, SM99]

The *ABC* forms the heart of our test environment, particularly the components *Test Case Design*, *Execution Engine*, and *Verification Engine* are based on existing *ABC* modules. The *ABC* bases on an interpreter kernel, which can be extended by specialized modules at runtime. In the next section we first discuss the key aspects of the *ABC* interpreter, before we sketch the used *ABC* modules.

4.2.1 The High-Level Language Interpreter

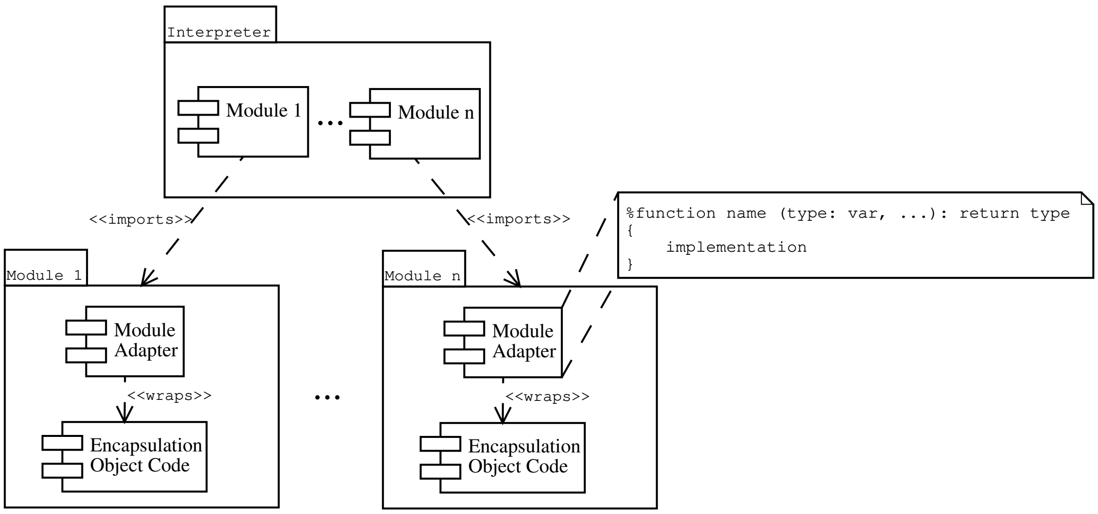


Figure 4.4: The High-Level Language Interpreter

The *ABC* provides a flexible interpreter, capable of interpreting a simple, pascal-like, imperative language, called *High-Level Language* (*HLL*) [Cla97, Hol99a]. The interpreter can be dynamically extended by new data types and functions, where the additional types and functions are provided by *METAFrame Modules* that can be imported into the interpreter core. Such modules consist of:

— *Encapsulation Object Code*, implementing the actual functionality and corresponding types. This code is given as C++ classes.

— *Module Adapter*, wrapping the *Encapsulation Object Code* into HLL-functions and types accessed by the interpreter.

A module adapter is particularly responsible for wrapping and unwrapping HLL data objects into C++ objects and also maps the execution of a HLL function to the corresponding encapsulation code.

For the integration of a *METAFrame Module* into the interpreter, a special compiler is used to translate the *Module Adapter*, specified in a certain format suitable for extending the interpreter (cf. Figure 4.4), into C++ source code which can then be compiled and linked together with the *Encapsulation Object Code* into a concrete *METAFrame Module*. These modules are built as shared libraries so that they can be imported at runtime into the interpreter. For a more technical discussion of the implementation of modules and their integration into the interpreter, please refer to [Hol97, Hol99b].

Next we will present a rough overview of special modules that are already available within the *ABC*, and fit into the *ITE* approach particularly well. Note that we will discuss the usage of some of the modules in more technical details when we present their test specific adaptations later on.

4.2.2 Polymorphic Labelled Graphs

One of the most important modules within the *ABC* is the *Polymorphic Labelled Graph Library* (*PLGraph*) [vdBBC+97, Hol98], which provides a flexible graph data structure. The main characteristic of these graphs is that all graph components (i.e. the graph itself, the nodes, and the edges) can store information via labels. This labelling concept is *polymorphic* in the sense that each graph component can contain more than one label. The labels themselves are implemented as C++ classes that are inherited from a common label class. Labels can contain various information and have access to the graph structure as well as to the corresponding components. In particular the ability to display graphs has been implemented by labels. Furthermore, special interfaces exist that ensure that the content of the labels can be saved and reloaded, or will be copied during the cloning of a graph.

An important extension of the generic graph library is called *Service Logic Graph* [SM99]. Here a special description is attached to each node, consisting, beyond others, of a name, a parameterization, and a portion of HLL code. In the context of service logic graphs, the nodes are called *Service Independent Building Blocks* (*SIB*). These service logic graphs are especially important in forming the basis for our test cases, cf. Section 4.4. Furthermore, for service logic graphs there exist a graph editor, used for the graphical construction of service logic graphs.

4.2.3 Tracer

A special module, called `Tracer`, can be used for the execution of a service logic graph. For each node executable code has to be implemented in HLL. Within this code, called *Run-Time-Code* (*RTC*), the call to the functionality specified in HLL can be implemented. The tracer executes the RTC code of the nodes, starting with the unique start node of a service logic graph. The tracer will then run in a loop until it reaches an end node:

1. Call the RTC code of the actual node.
2. The RTC code calls the tracer back, giving the identifier of an outgoing branch.
3. If the tracer finds an invalid branch, i.e. the branch given is not a valid branch for this node, it terminates the execution with an error.
4. The tracer waits a specified period of time, and then executes the RTC code for the node, which is determined through the branch.

As the RTC code is specified in HLL it is possible to use all the functionality provided by the used modules, and particularly all control structures that are defined in HLL (`if-then-else` and `while`).

4.2.4 Verification Facilities

Service logic graphs are subject to local and global verification. Local constraints are specified as *assertions* in HLL. A special module `LocalCheck` provides several methods which ease the specification of these constraints. When performing a local check, for all nodes of a graph it will be checked whether they fulfill their constraints. For checking global constraints the *ABC* consists of a game based model checker for the modal μ-calculus [Yoo03, SCK$^+$95]. Here the atomic propositions are attached as node- resp. edge-labels and can then be accessed from within the model checker.

Before using this model checker for checking *ESLTL*-formulae, a preprocess, as described in Section 3.4.3, will be performed to compute the atomic propositions and to transform the *ESLTL*-formula. The constraints are organized in libraries, and can be attached to a graph at need. This relationship of a graph and the corresponding constraints is stored within a graph.

4.3 An Architecture for an Integrated Test Environment

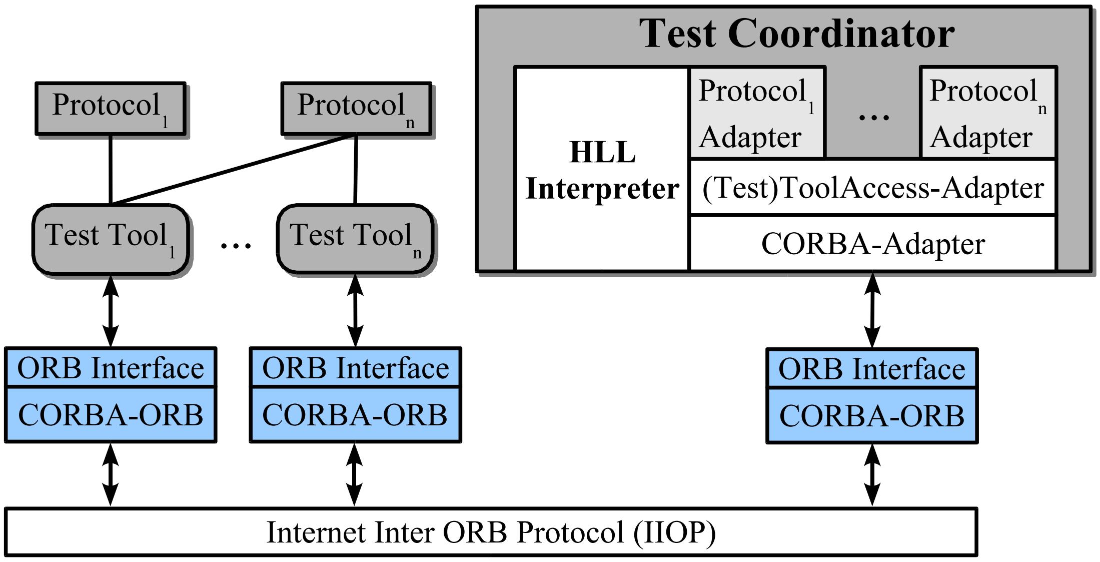


Figure 4.5: Communication Architecture of the Integrated Test Environment

As presented in Section 4.1 the *ITE* consists of a framework portion (*Test Coordinator*) that is completed by concrete test interfaces which allow us to test a certain test scenario. Figure 4.5 shows the communication architecture of the integrated test environment in more detail. Each test tool supports one or more test interfaces (*PO* or *PCO*) to the system under test (see Section 4.3.1 for a detailed discussion of test tools). In the *ITE* we call the set of stimuli and observation points for a certain test interface *Test Protocol*, or simply *Protocol* when unambiguous from the context. A protocol adapter wraps the functionality[1] provided by a test protocol, and makes it available to the test coordinator, or more precisely to the HLL interpreter.

[1]Usually in the testing context we are only interested in defining new functionality rather than defining new types.

The availability of a test tool depends on the circumstances at runtime. Therefore, the `TestToolAccess` adapter in cooperation with the `ToolAccess` adapter is responsible for administrating the used test tools in a registry. For the implementation of the communication layer between the *ITE* and the test tools (cf. Figure 4.1) we use CORBA. CORBA is the acronym for **C**ommon **O**bject **R**equest **B**roker **A**rchitecture, a vendor-independent architecture and infrastructure that computer applications use to work together over networks. Using the standard protocol **I**nternet **I**nter **O**RB **P**rotocol (*IIOP*), a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network. CORBA is an established industrial standard for complex middle ware solutions [Obj99]. Within the *ITE* a special adapter is responsible for encapsulating the general CORBA functionality.
The test tools offer their functionality via an interface description, specified in the **I**nterface **D**efinition **L**anguage (*IDL*)[2], which can then be accessed from within the *ITE*.

In the next section we will first discuss what a test tool is in the context of the *ITE*, before we discuss the interaction between the *ITE* and the test tools in more detail.

4.3.1 Test Tool

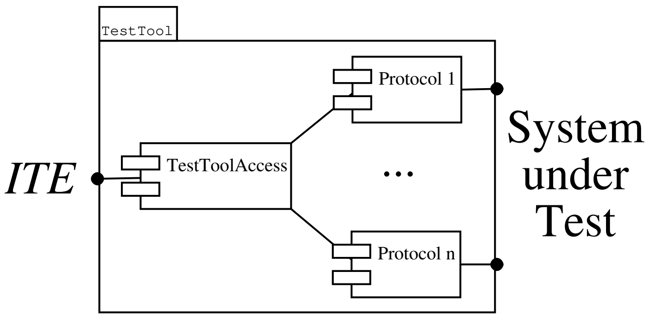


Figure 4.6: Test Tool

In general a test tool within the *ITE* is able to observe and stimulate the system under test via several *PO* or *PCO*, which we also call *Test Interfaces*. This capability of a test tool is pictured in Figure 4.6. A test tool is made up of a control interface (`TestToolAccess`), to be remotely accessed from the *ITE*, and several test protocols (`Protocol`$_1$ ... `Protocol`$_n$), each responsible for testing a certain test interface of the

[2]Note that in the following we will often refer to the interface description as IDL.

system under test. Sometimes it is even possible to apply a certain test protocol via several system interfaces, e.g. communication protocols exist that can be transported over TCP/IP as well as packed in HDLC frames of ISDN (cf. Section 6.1). So a test tool in terms of the *ITE* establishes the connection between a remote interface and one or more test interfaces. One of the advantages of treating test tools in such a general way is that one is not restricted to implementing a certain protocol within several test tools. So we can ensure that one is, in principle, not limited by the weaknesses of specific test tools, but can use for every test task, e.g. a particular test stimulus, always the best available test tool. Note that from the conceptual point of view this approach works quite well, as it is ensured from within the *ITE* that the involved test tools does not interfere with each other. This is because of the synchronous execution of test cases: Only one test action is executed through a particular test tool at one moment.

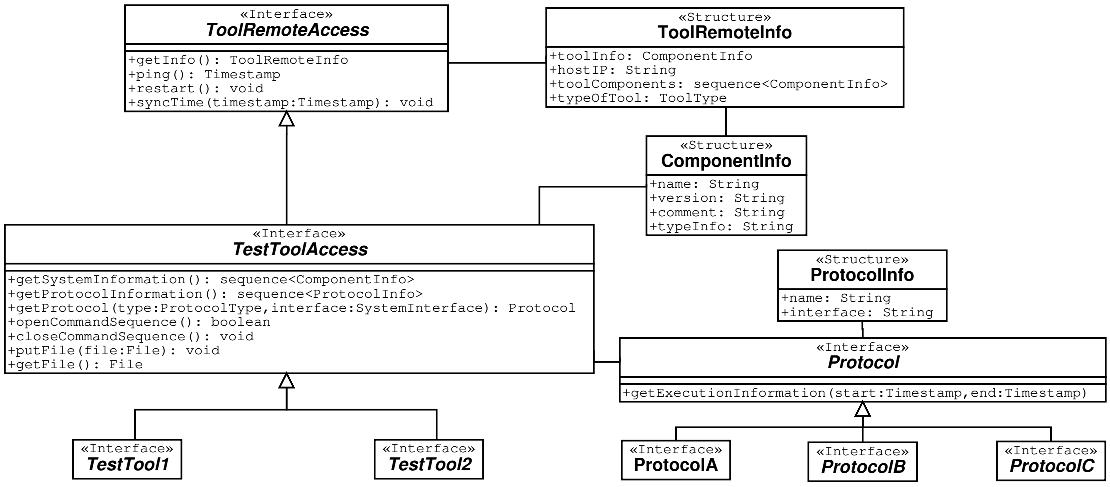


Figure 4.7: Tool Remote Access

Let's focus on the technical realization of a test tool, cf. Figure 4.7. The general interface `ToolRemoteAccess` is particularly suited to providing functionality for requesting detailed information about a (general) tool (`getInfo`) and for checking its availability (`ping`) resp. restarting the tool (`restart`). The information about a tool is gathered within the structure `ToolRemoteInfo`, which provides information about the tool itself, its IP address, the type of the tool, and detailed information about the components the tool is built on. Information about the tool and its subcomponents are collected in the structure `ComponentInfo`, where the *name*, the *version*, the information about the *type* of the tool, and a detailed *comment* is stored. This information is needed in every tool coordination scenario and therefore it is not restricted to the *ITE*. To take the specialities of the test tools into account, a specialization of the interface `ToolRemoteAccess` is provided by `TestToolAccess`. Here additional information about the tested subsystem (`getSystemInformation`) and about the supported test protocols (`getProtocolInformation`) is available. The information about a protocol (`ProtocolInfo`) consists of a *name* and the *interface*. Furthermore, a certain protocol on a specific system interface can be requested (`getProtocol`) if such exists. In various communication protocols the stimuli have to be submitted during a stipulated time slot. This cannot be ensured from within the *ITE* when the stimuli are distributed over several test blocks, because of the indirect step that is introduced by means of the CORBA communication in between. Therefore it is possible to send sequences of commands to a test tool that are executed as a whole. Two methods are responsible for dealing with that: `openCommandSequence` and `closeCommandSequence`. All commands that are dropped in between are interpreted as a sequence that is executed immediately after the closing of the command sequence. Note, however, that these commands can take place on different supported protocols. Finally a test tool supports the generic transfer of files in both directions through `putFile` and `getFile`.

A concrete test tool has to provide an access by specializing the interface `TestToolAccess` and one or more protocols, which provide the real testing facilities, by specializing the interface `Protocol`. The only method prescribed by this interface is `getExecutionInformation`, which retrieves the start and end timestamp of the last executed command.

4.3.2 Interaction between the Integrated Test Environment and its Test Tools

Figure 4.8 shows the cooperation between the test tool access adapter and a test tool. The relationship between the two of them will be established loosely, i.e. the test tool registers itself at the *CORBA Nameservice*, a standard CORBA service which allows clients to find objects based on names instead of their **I**nter **O**bject **R**eference (IOR) or their CORBA location[3]. The *ITE* resolves tools that are available within the nameservice, and binds them in a local tool registry. In that way we avoid the lookup of the tool via CORBA for every communication, i.e. call of a method. A local test tool instance is especially created for every remote tool and it serves as a facade to the remote test tool. This tool registry is maintained by the, more general, `ToolAccess` adapter. The more specialized `TestToolAccess` adapter uses the functionality by delegation, i.e. because in the adapter concept no inheritance is possible. Furthermore, the `TestToolAccess` adapter adds functionality to support the specialities that have to be considered when dealing with test tools as a special instance of general tools.

[3]A CORBA location, or *corbaloc*, is a URL format object reference.

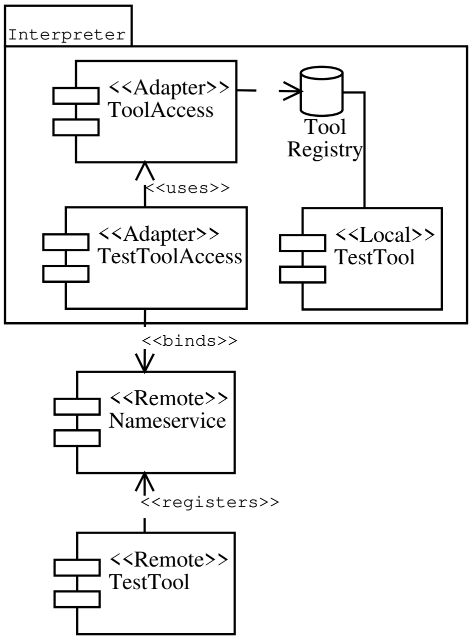


Figure 4.8: Cooperation between the *ITE* and a Test tool

More technically the `ToolAccess` adapter is responsible for the general management of (remote) tools. It maintains a tool registry, where the local facades for the remote tools are stored. To allow convenient usage of the test tools, the `ToolAccess` adapter provides access to the general tool specific methods (see Figure 4.7) via the tool's identifer, used for storage in the registry. For the management of the tool registry it provides methods e.g. for retrieving the number of registered tools, or for testing whether a certain tool identifier is already in use. By using a tool identifier one is able to check whether a tool is alive, to restart a tool, and to retrieve all the meta information that is provided by a tool, i.e. name, IP address, comment, version information, and the detailed component information. Furthermore, it maps the exceptions that can occur during the usage of a tool, to special types of HLL (`TAException` and `CorbaException`), and provides functionality to check, e.g. whether a communication problem arises (CORBA exception). This is needed, because no concept of exceptions exists in HLL.

The `TestToolAccess` adapter uses the functionality of the `ToolAccess` adapter for the general management of the tool registry. The most important additional functionality is the registration of new test tools. Here a new facade for a test tool will be created and later bound with a unique tool identifier to the tool registry. Furthermore, it provides methods for accessing the specialized methods of test tools, i.e. handling of command sequences, retrieval of system under test information, and protocol information handling via the test tool identifier.

With this concept we are able to fully abstract ourselves from the underlying communication layer, so that within the interpreter test tools can be accessed via the tool registry. Changing the communication layer would therefore only affect the `ToolAccess` adapter.

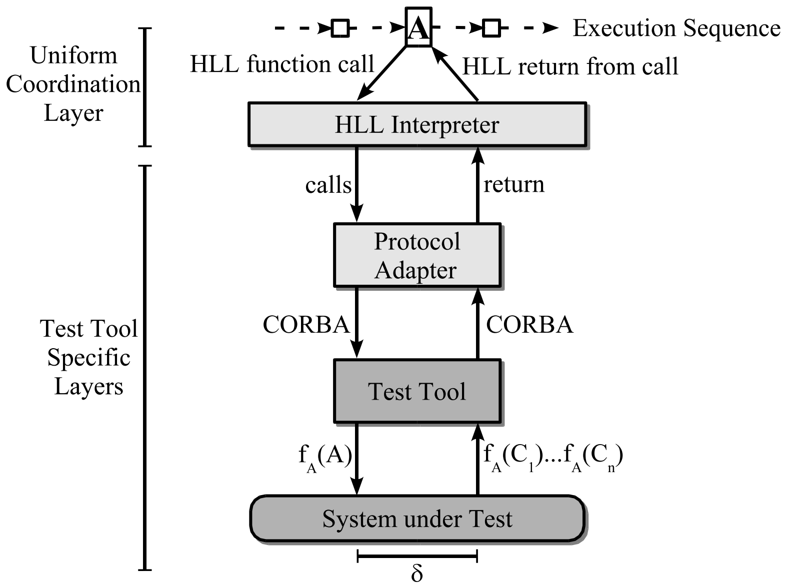


Figure 4.9: Test Tool Coordination

Figure 4.9 presents the conceptual realization of the execution semantics for test cases, as presented in Section 3.4, within the *ITE*. Let us assume that the test block $\boxed{\text{A}}$ should be executed. This results in a corresponding HLL function call that will be processed by the HLL interpreter. The test protocol adapter that defines the HLL function communicates with its corresponding test tool. This first translates the test block into the real stimulus ($f_A(A)$) and then performs the stimulation of the system under test. Within a well-defined time period δ, the test tool gathers all resulting responses of the system ($f_A(C_1), \ldots, f_A(C_n)$) and stores them for further processing. This agrees with the structure of the corresponding test graph, which represents the execution semantics of the test case (cf. Section 3.4.2). Here at every node of a test graph all the outputs from the system (responses) are collected by a reflexive edge that is labelled with the complete input alphabet. This loop can only be exited via the special timeout edge (δ). Note that in the real implementation the timeout depends either on the test tool or even on a specific command. The test tool, however, must ensure that it is fixed within a certain situation to guarantee a deterministic, and therefore repeatable, execution of test cases. After the execution of the stimulus and the expiration of the timeout, the test tool signals the correct execution to the protocol adapter, or else the error cause. The corresponding methods of the protocol adapter then terminate as well, so that the test block execution ends. In case of check test blocks, the results of this execution lead to a selection that decides which test block has to be executed next.

The coordination task can be split into two different layers: a *Uniform Coordination Layer* and a *Test Tool Specific Layer* (see Figure 4.9). The first one assures that test cases will be executed from within the *ITE* in a uniform fashion, independent of specific characteristics of the considered testing scenario. The test tool specific layers are then responsible for the concrete implementation of the test blocks which depend on the underlying test tools and their test protocols.

4.4 Realization of the Fundamental Functionalities of the *ITE*

Whereas in Section 3.3.1 the requirements of the *ITE* were discussed, within this section we present their concrete realization.

4.4.1 Test Case Design

As mentioned above, there exists an extension of *ABC*'s generic graph library, called *Service Logic Graph*, that is well suited to the specification of test cases, as stated in Definition 3.2. In our context we denote the *SIB's* as *test blocks* and our test cases are then special service logic graphs.

A test block, as the elementary entity of a test case, is defined by three different specifications:

1. Specification of its *interface*, consisting of a name, a class, a list of formal parameters, and a list of outgoing branches,
2. specification of its *execution code*, used for test execution and defined in HLL (cf. Section 4.4.3), and
3. optional specification of *local check code*, used for local consistency checks, e.g. concerning the correct parameterization (cf. Section 4.4.2).

Figure 4.10 shows an example of a test block interface[4]. First, the name and the class of the test block are defined. The class is mainly needed to support comfortable browsing through all available test blocks afterwards, but also for the partition of the test blocks into external and internal ones. The parameters of a test block are

[4]Note that all keywords are uppercase.

```
SIB name
CLS class
PAR parName1 NUM min max default
PAR parName2 STR maxlen "default"
PAR OPT parName3 SEL "Sel1" "Sel2" ... "Seln" END i
BR default
```

Figure 4.10: Example of a Test Block Interface File

specified by the keyword `PAR`. Parameters consists of a name, a type (`NUM` for integer values, `STR` for strings, and `SEL` for an enumeration of strings), and depending on their type, additional information:

— For `NUM` the range and a default value can be specified (`min`, `max`, `default`),

— for `STR` the maximal length of a string (`maxlen`) and a default value can be specified, and

— for `SEL` first the alternatives are specified, ending with keyword `END`, and additionally the index `i` to the default value is given. Note that $1 \leq \mathtt{i} \leq n$.

Note that according to this declaration the parameter domains are always finite. Furthermore, each parameter can be declared as optional by the keyword `OPT`, cf. the declaration of the last parameter in Figure 4.10. We will use this keyword to distinguish between internal and external parameters. Internal parameters are only needed for environmental purposes, e.g. for specifying the test tool which should execute the command. External parameters will be transferred to the actual test tool.

Finally, a test block interface makes up the definition of the outgoing branches. As stated in Definition 3.2 we only distinguish between two cases, i.e. action test blocks with one outgoing branch (`default`), and check test blocks with two outgoing branches (`true` and `false`). Please note that these test block specifications were usually generated automatically and do not have to be edited by hand, cf. Section 4.5.

Figure 4.11 illustrates the graphical definition of a test case. For test case design an editor is available that allows a test engineer to add test blocks to a test case, to connect their instances by edges, and finally to set their internal parameters. This high level of abstraction is tailored for an easy design of test cases, where no particularly programming expertise or detailed knowledge of the test tools and their coordination is needed for the design task. The test case design is illustrated according to a concrete example in Chapter 5.

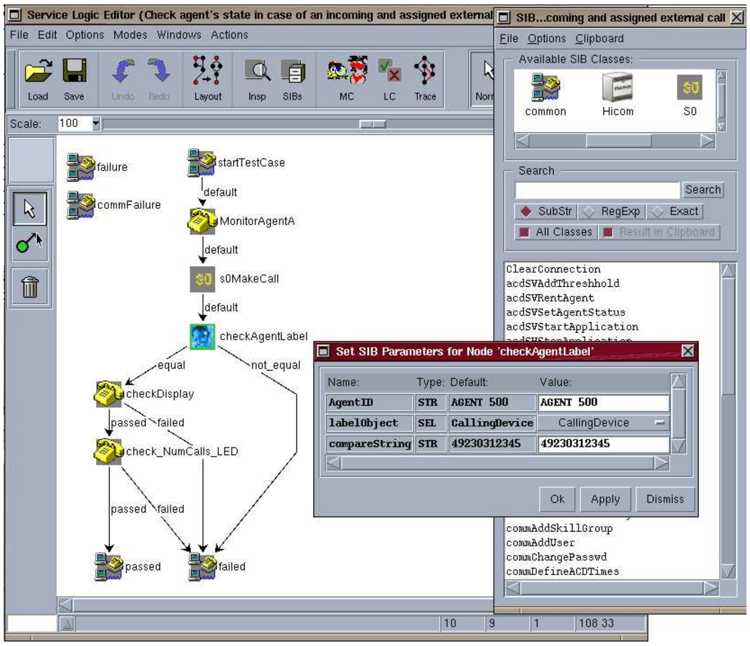


Figure 4.11: Specification of a test case through the *ITE*

Reuse by Hierarchical Test Case Design

Test cases have recurrent structures, e.g. when testing complex systems usually some applications must be started during an initialization phase or a user must log on to the system. Hierarchical test case design allows test engineers to specify these tasks as test cases once and for all and to make them available as generic test blocks (or macros) for later reuse. This accelerates the design of new test cases and it supports the maintenance of the test suites: modified macros automatically update all their instances. The *ITE* supports a truly hierarchical design, where macros are allowed to make full use of other already existing macros. Figure 4.12 shows how this works in practice (cf. [SMBK97] for a detailed discussion): Within an abstraction step a (part of a) test case is stored as a macro, which becomes directly available as a new generic test block. In addition to the identifier and the name of the macro, the formal parameters and the outgoing branches have to be specified. The parameters of the macro can be mapped to (selected) parameters of the underlying test blocks. E.g. in Figure 4.12 the parameter P_{12} of test block TB_1 and P_{21} of test block TB_2 become parameters of the macro. Similarly, the set of (unset) outgoing branches of the underlying test blocks defines the outgoing branches of the macro. E.g. in Figure 4.12 the unset branches of test block TB_2 and TB_4 (denoted through the dashed edges) will constitute the outgoing branches of the macro. As usual, the resulting hierarchy is a design and management aid without

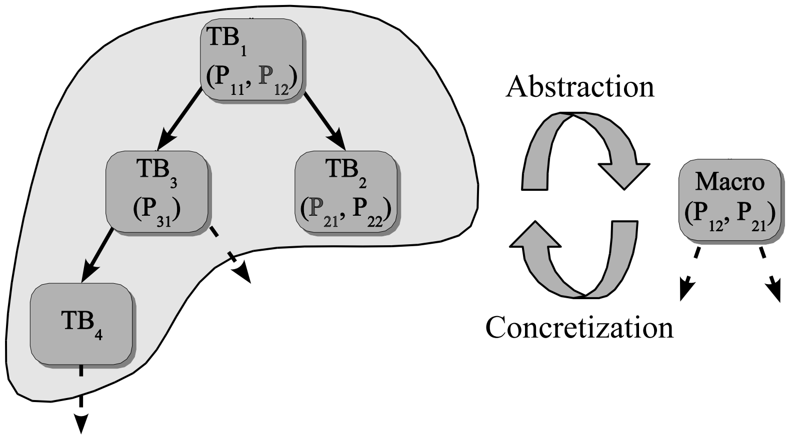


Figure 4.12: Hierarchical Test Case Design using Macros

any influence on the execution time: during the execution macros are automatically unfolded (*concretized*) and the underlying graph structure will be executed.

4.4.2 Test Case Verification

Test cases are subject to local and global constraints, cf. Section 3.4.3. To support a reliable test case design, however, both of them are needed.

Local check code, i.e. assertions for test blocks, can be bound to single test blocks, whole classes of test blocks, and can furthermore be specified globally. Depending on the scope, the local check code is stored in different files, i.e. test block local check code is stored in a file like `<TestBlockName>.lcc`, class local check code under `<ClassName>.class.lcc`, and global local check code at `_global.lcc`. This reduces redundancies in the local check code, as we are able to define the check of e.g. parameters common to a whole class only once.

```
var String:  toolName =
    SD.getSibParameter (SDLocalCheck.local_check_node, "toolName");
if (empty (toolName)) then
    SDLocalCheck.localCheckError ("Unset Tool Name");
fi;
```

Figure 4.13: Example for Local Check Code

Figure 4.13 shows a portion of local check code. During this process the `Local-Check` module provides access to the currently checked node of the graph through

`SDLocalCheck.local_check_node`. First the value of the parameter `toolName`[5] is read out and assigned to a local variable. Afterwards it is checked whether this value is empty or not. If so, a local check error will be raised, which usually results in an error message being presented to the user, together with highlighting of the erroneous node.

Furthermore, by local checking it can be assured that all edges in the test case are labelled correctly. More technical details about writing local check code can be found in [MET99].

For global checking first of all global constraints have to be specified in a temporal logic (*ESLTL*), so that the model checker can verify whether a test case fulfil certain properties. An open issue is the realization of the pattern system proposed in Section 3.4.4 that supports test engineer during the specification of the global constraints. Here a flexible management of the underlying data is needed. On the one hand it should support an automatic and efficient generation of concrete constraints that can be further processed by the model checker. On the other hand it must offer enough information for building a corresponding graphical user interface (*Constraint Editor*) for the specification of constraints. For this purpose we use the *Extensible Markup Language* (*XML*) [Wora] and *Document Type Descriptions* (*DTD*). XML is a simple flexible text format derived from the *Standard Generalized Markup Language* (*SGML*) [ISO86]. In principle a DTD is a grammar which prescribes how correct XML documents of a specific type have to be constructed. Thereby an XML based notation concentrates on the information itself and its structure rather than on its presentation. In addition there is a variety of powerful tools and libraries for an automatic processing of XML based documents.

For an implementation of our pattern system we propose three different DTD's, one for each level of the pattern system, cf. Figure 3.11:

1. A DTD for storing a set of logic patterns together with their mapping to a concrete logic in a given syntax (here *ESLTL* in a syntax suitable for the built-in model checker) (cf. Figure 4.14).

2. A DTD for storing a composite pattern (cf. Figure 4.15), which consists of two parts:

   (i) Definition of the used placeholders together with the type of their corresponding input dialog, used for their specification.

[5] Note that every external action and check test block needs this parameter to resolve the test tool that is responsible for its execution.

(ii) Definition how to generate the corresponding logic pattern, i.e. mapping of the (global) placeholders of the composite pattern to the (local) placeholders of each logic pattern.

3. A DTD for storing concrete constraints, i.e. the reference to the corresponding composite pattern, together with the concrete values for the placeholders.

While the complete DTD's can be found in appendix B.1 we illustrate their usage by means of a concrete example, i.e. the definition of the composite pattern of example 3.4.

```
<pattern name="response" scope="global">
    <raw> <![CDATA[AG_F(']]> </raw>
    <p/>
    <raw> <![CDATA[ => AF_F(']]> </raw>
    <s/>
    <raw> <![CDATA[))]]> </raw>
</pattern>
```

Figure 4.14: Mapping of the Logic Pattern *Response* with Scope *Global* to *ESLTL*

First we have to define the mapping of the corresponding logic pattern to the concrete syntax of *ESLTL*. In Figure 4.14 the mapping of the logic pattern *Response* with scope *Global* is shown. Basically what is stated here is the following: `AG_F ('P => AF_F ('S))`, all other symbols are just representing the structure of the document.

In Figure 4.15 a shortened version of the representation of the composite pattern of example 3.4 is presented. Beside the meta information about the new pattern, i.e. its *name*, informal *description*, and a corresponding *class* used for a classification of the composite patterns, the XML document can be split into two main sections. The first section (`input`) prescribes which *steps* are needed to gather all required information about how to fill the placeholders in the corresponding logic patterns. For each placeholder a matching step has to be defined, which consists of a *number* to be identified later, and an informal description that is presented to a user, together with the declaration of a dialog. Currently there are two different dialogs supported, one for specifying test blocks with parameters (`SelectTBWithParameters`), and one for specifying them without parameters (`SelectTB`). The set of available dialogs, as well as their attributes, is not prescribed within the DTD. The main reason for doing this is to offer enough flexibility and to allow to create new, specialized dialogs in the future. One can think of additional dialogs, e.g. to specify whole subformulae.

```
<CompositePattern name="ResourceBalance" class="Resource">
  <patternDescription> ... </patternDescription>

  <input>
    <step no="1">
      <stepDescription> Selection of the first test block </stepDescription>
      <dialog> SelectTBWithParameters </dialog>
    </step>
    <step no="2">
      <stepDescription> Selection of the first test block </stepDescription>
      <dialog> SelectTBWithParameters </dialog>
    </step>
  </input>

  <generateLogicPattern>
    <logicPattern pattern="response" scope="global" ...>
      <logicPatternDescription>...</logicPatternDescription>
      <sibs>
        <p>
          <name> <advancedLink step="2"> nameSib </advancedLink> </name>
          <args> <simpleLink step="2"/> </args>
        </p>
        <r>
          <name> <advancedLink step="1"> nameSib </advancedLink> </name>
          <args> <simpleLink step="1"/> </args>
        </r>
      </sibs>
    </logicPattern>
    <logicPattern type=...>
      ...
    </logicPattern>
  </generateLogicPattern>

</CompositePattern>
```

Figure 4.15: Representation of the Composite Pattern of Example 3.4

The drawback of this flexibility is that it is possible to obtain syntactically correct XML documents that cannot be further processed, because of missing dialogs or incorrect links to their elements.
The second part defines the mapping of the inputs of the composite pattern to the placeholders of the logic pattern (`generateLogicPattern`). For each corresponding logic pattern an entry is provided (`logicPattern`) which is made up of the logic pattern name and its scope. Furthermore, an informal description has to be given. Then for each placeholder an entry needs to be specified which defines within which step the information can be found. The names of the placeholders are defined in the logic pattern definition. The names that are used for the generation will

```
beginenv;
    ...
    var (Bool * TAException):  rt;
    var String:  toolId := SD.getSibParameter (Tracer.current_node, "toolName");
    ...
    rt := ExampleProtocol.execCommand (toolId, ...);
    if (ToolAccess.isException (#2.(rt)) == false) then
       if (#1.(rt) == true) then
          Tracer.setBranch ("true");
       else
          Tracer.setBranch ("false");
       fi;
    else
       ...
       Tracer.setNode (FAILURE);
    fi;
    ...
endenv;
```

Figure 4.16: Example for the *Run-Time-Code* of a Check Test Block

then be matched. For the establishment of the relation between the information, given through the input steps and the concrete placeholders, two kinds of links are provided: *Simple Links* and *Advance Links*.

Simple Link If parameters for a test block have been defined by a `SelectTB-WithParameter` dialog, this link retrieves all name/content pairs of this step. Consequently a simple link is used when specifying test block parameters, cf. Figure 4.15.

Advance Link An advance link can be used to access a specific attribute of the corresponding dialog. In Figure 4.15 it is used to specify the name of the test block.

A concrete example of the use of this approach for the specification of constraints will be illustrated in Chapter 5.

4.4.3 Test Case Execution

Test cases can be executed in the *ITE* by means of the *ABC*-`Tracer` module. For this purpose for each test case corresponding RTC code has to be implemented. In Figure 4.16 an example of the RTC of a check test block is depicted. Note that only the relevant parts are shown here, whereas all others are suppressed. The RTC code is encapsulated in a local environment to avoid name clashes between different test blocks. First a variable is declared for storing the result of the command execution (`rt`): this consists of a boolean value and an exception. After that the value of a certain test block parameter is read out, where the reference to the currently active node is provided by the tracer (`Tracer.current_node`). Now the command can be executed in the context of a specific protocol, here `ExampleProtocol`, where both the tool identifier has to be given as well as well as all the relevant parameters needed for a correct execution of the remote method. After the execution one must first check whether general problems occurred during execution (communication failures or problems with the test tool resp. system under test). Here `#2.(rt)` denotes the projection to the second component of `rt`. If there are problems, we directly instruct the tracer to proceed with a dedicated error sink (`Tracer.setNode (FAILURE)`). Otherwise we are finally able to evaluate the real result and to proceed with either the `true` or `false` branch (e.g. `Tracer.setBranch ("true")`). Note that the RTC code can be generated automatically in most cases, cf. Section 4.5.2.

The detailed communication that takes place when a test block is executed can be seen in Figure 4.17. The call to `execCommand` within a test block is invoked at the corresponding protocol adapter. Here the two kinds of parameters are needed: the "normal" parameter are used to e.g. determine the concrete tool, whereas the optional parameters are transferred to the real (remote) test tool (here denoted through `params`). The protocol adapter first resolves the local test tool via its unique tool identifier, which is given as a parameter, from the `ToolManager`. The `ToolManager` is available as a singleton. The local test tool serves as a facade to the remote test tool, which provides the real functionality of the corresponding test protocol. With this information, i.e. the local test tool facade, together with the information about the protocol (available from the protocol adapter itself) and eventually additional information about the considered system interface, the concrete (remote) test protocol instance is retrieved by the local test tool, via the remote test tool. Before the command can really be executed it has to be registered at the `CommandManager`. This is needed to allow subsequent check test blocks to refer to prior actions via a unique command identifier. The `CommandManager` stores the command identifiers as well as the start and the end time of their execution. Finally after the registration of the command it can be executed via the concrete remote test protocol. At the end of the execution the protocol adapter informs the `CommandManager` of the termination of the command, and returns itself.

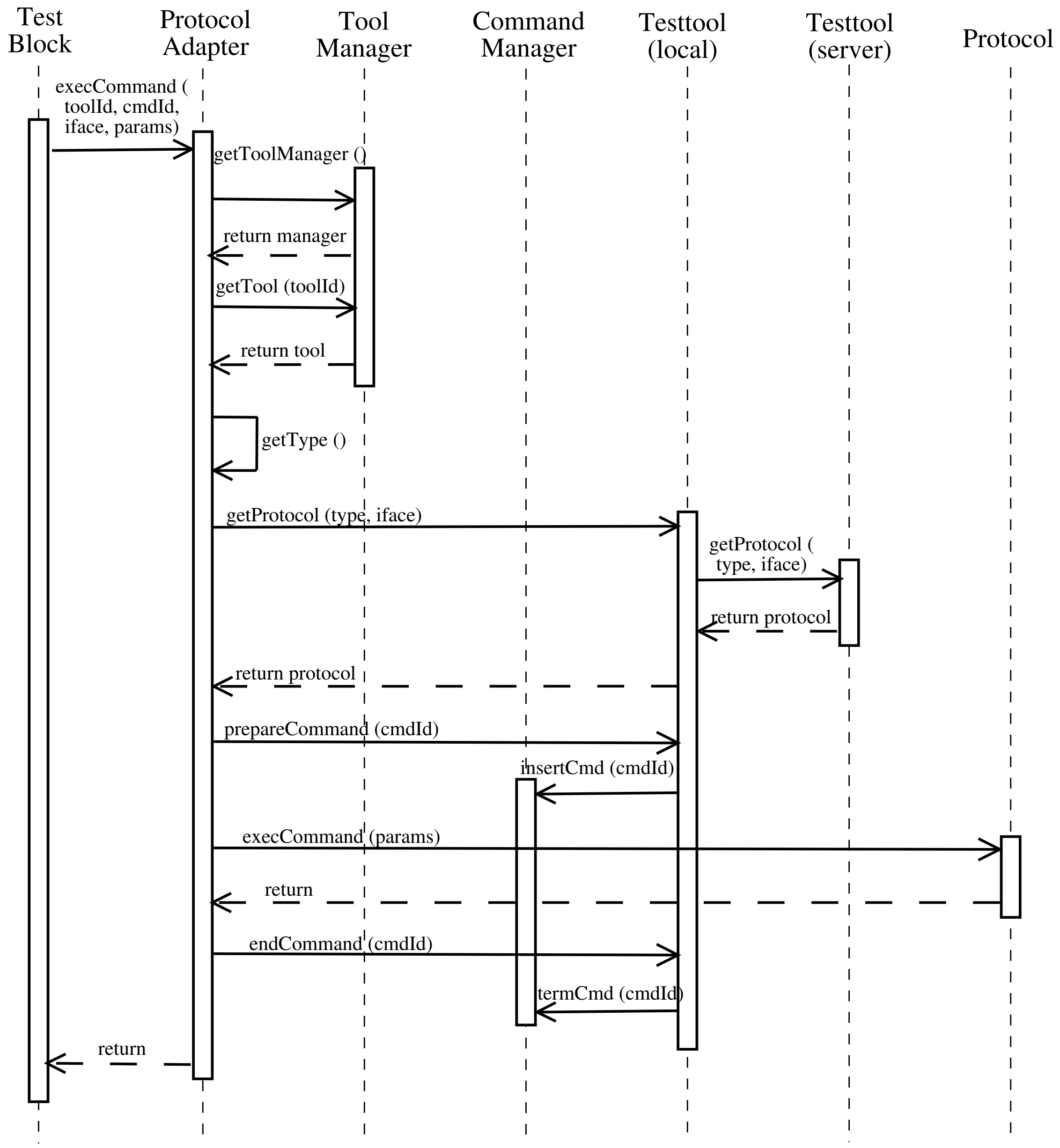


Figure 4.17: Command execution

4.4.4 Test Case Analysis

During the test case execution and in cooperation with the other *ITE* components the test coordinator gathers detailed information and prepares it for a test case analysis in the form of a test report. The basis of the test reports is an XML DTD, cf. appendix B.2. The key reason for a test report is to collect characterizing information about the system under test (configuration, version, ...), the involved test tools (name, version, used protocols, ...), and the results of the real test execution (processed test blocks with parameterization, execution time, ...). All this data is important for a sophisticated analysis of the test runs afterwards, and furthermore ensures repeatable tests, as it provides enough information to set up the test scenario properly. In general a test report summarizes the execution information about a whole test suite.

The module `ReportGenerator` provides the needed functionality for the generation of a test report. So basically, specific internal test blocks are needed for the general handling of test reports and the RTC code of (external) test blocks has to be further annotated so that a suitable test report can be generated. Usually, all the meta data that cannot be gathered from within a test case itself must be requested from the test engineer before test execution, e.g. a detailed description of the considered test setting. In the following we discuss the functionality provided by the `ReportGenerator` module in more detail:

openReport Creates a new report and opens it for processing.

closeReport Closes a report.

setScenario Allows the naming of a scenario.

describeTestTool Describes a test tool. Here all relevant data about a test tool can be logged, e.g. its name, version, IP address, configuration, etc. Note that this information is usually already provided by the involved test tools and can be retrieved by `getInfo`.

describeSystem Describes the system under test. Again all relevant data can be logged, e.g. names and versions of the subsystems, further configuration information, etc. This information is also available by the corresponding test tool (`getSystemInformation`).

startTestCase Begins a new test case section in the test report.

addLogEntry Allows the logging of various data in a test report.

logTestBlockData Logs the execution of a test block. Here not only the time-stamp and the name of the test block is logged, but, more important, the concrete parameterization and the default one. Eventually, when connected to a version control system, the version number is logged.

logBranch Logs the chosen branch during execution. Together with the information of the executed test blocks this defines the complete execution sequence.

addErrorMsg Specific error messages can be logged, e.g. the processing of the test block `Failed`.

Taken together a test report provides enough information to present either a compressed overview about the execution of a test suite, i.e. in particular whether the whole test suite evaluates to passed or failed, or an elaborated test execution log, where detailed information about the execution of a single test case is given. The *ITE* manages the test reports by a special internet service that is able to provide role-based access to test reports, together with the presentation of test reports at different levels of detail, suitable for each role. E.g. guests will only see a compressed overview of selected test suites, whereas test engineers are allowed to see full test reports. The usage of this internet service will be presented in Chapter 5.

4.5 Test Tool Integration

This section discusses how new test tools can be integrated into the *ITE*. There are three main tasks that have to be carried out during the integration of a new test tool:

1. Definition of an IDL for the remote access of the test tool and at least one IDL that specifies a test protocol.
2. Implementation of the proposed functionality in the test tool so that it can be exported via CORBA to the *ITE*.
3. Based on the offered IDL the functionality of the test tool must be integrated into the *ITE* so that it is accessible within the interpreter.

The definition of an appropriate IDL depends on a concrete test setting as well as on the abilities of a test tool, and can therefore not be solved in advance. Assuming that this IDL already exists, the other two tasks will be discussed in more detail below. Concrete examples for the definition of appropriate IDL's can be found in Section 4.6.1 and Chapter 5.

4.5.1 Implementation of a CORBA Interface in a Test Tool

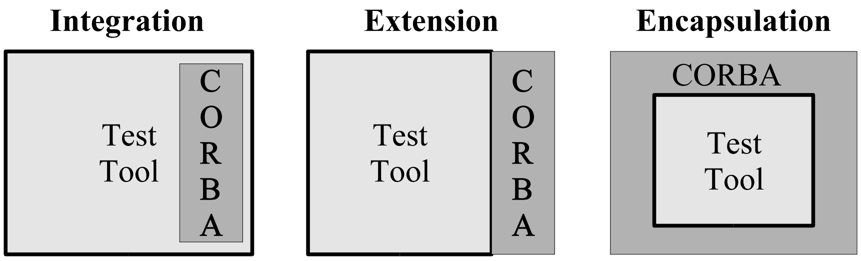


Figure 4.18: Variants for the Implementation of a CORBA Interface in a Test Tool

The implementation of the CORBA interface in a test tool can be done in at least three different variants, cf. Figure 4.18:

Integration The CORBA interface will be fully integrated into the test tool. This is only possible when the source code of the test tool is available.

Extension The CORBA interface will be implemented in a separate library. This library is responsible for the communication aspects as well as for controlling the test tool. One possibility is that the test tool initializes the library and afterwards calls a special method. This method transfers the execution flow to the external library so that it is able to control the test tool.

Encapsulation This approach is particularly well suited to command line test tools. Here a CORBA server will be implemented and it realizes the required functionality by delegating it to the test tool. For this purpose the test tool will be called, together with all necessary command line options and parameters, in a shell.

4.5.2 Integration of a CORBA Interface into the Integrated Test Environment

For the integration of a test tool into the *ITE*, the corresponding adapters have to be implemented. Furthermore, we need test blocks, together with RTC that provide access to the functionality during test case design. Most of the work, however, that has to be done during the integration of test tools into the *ITE*, can be automated. Figure 4.19 sketches the workflow of the automatic generation of adapter code as well as of the test blocks. An IDL describing the test protocol serves as the source of information for the transformation process. On the one hand the stubs for the IDL will be generated with a "normal" *IDL Compiler*, shipped with the CORBA

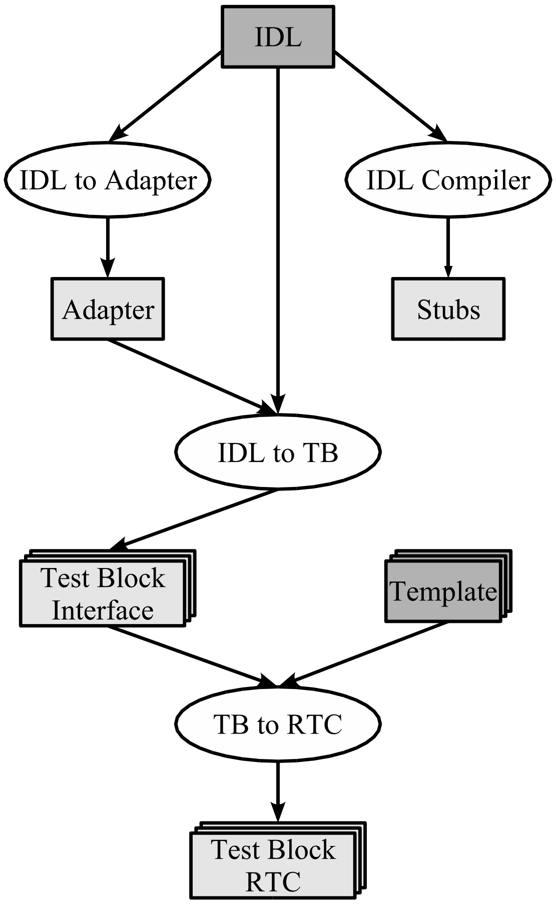


Figure 4.19: Workflow for the Automated Integration of Test Tools in the *ITE*

ORB. On the other hand a special *IDL-to-Adapter* compiler is used to create a corresponding protocol adapter, where for each method of the IDL a HLL function will be generated. The IDL together with the protocol adapter is also the source for creating test blocks with the *IDL-to-TB* compiler. For each function of the protocol adapter a matching test block will be created as well. The test blocks can then be used in the test case design. For the execution of test cases, RTC for the test blocks also needs to be generated. For this purpose the test block interface, together with special templates, forms the basis for creating its particular RTC by means of the *TB-to-RTC* compiler for each test block.

In the following we describe each of the used compilers in detail.

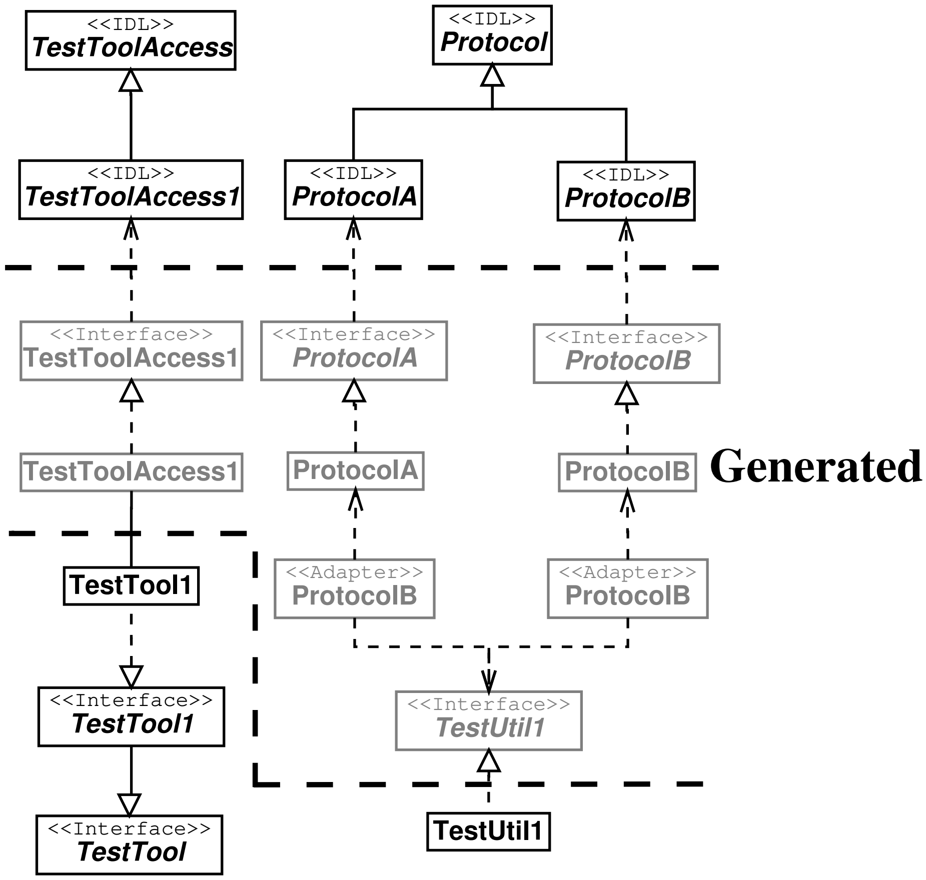


Figure 4.20: Tool Integration

IDL to Adapter Compiler

While the generation of the protocol adapter out of an IDL can be done automatically, the implementation of (some of) the encapsulation classes still has to be done manually. Figure 4.20 provides an overview showing which classes are needed to make the functionality of a test tool available within the *ITE*. The IDL's that the test tool offers are shown at the top. In this class diagram the test tool provides one access interface ($\texttt{TestToolAccess}_1$) and two test protocols ($\texttt{Protocol}_A$ and $\texttt{Protocol}_B$). The *idl* compiler generates the stubs to the IDL's. Furthermore the *idl-to-adapter* compiler generates a corresponding protocol adapter for each test protocol[6]. For each method provided by the IDL a corresponding HLL function will be created. The signature of the HLL functions will be enriched with respect to the original IDL method, by two additional parameters: `toolName` and `commandName`. These parameters are used to resolve the concrete test tool instance during the exe-

[6]Note that all code that is automatically generated is presented in light gray in Figure 4.20.

cution, cf. Section 4.4.3. In addition, two special classes are needed by the protocol adapter:

1. A local test tool class, acting as a facade for the remote test tool, and
2. a utility class for a test tool which helps to compute the results of check functions.

The local test tool class manages all accesses to the remote test tool class and is a specialization of the general `TestTool` class. It acts as a facade for the remote class and provides most of its functionality by delegating it to the remote class. Note that we view a facade in the sense of [GHJV95]. The advantage of this approach is that the tool registry of the *ITE* must only handle references to local instances, rather than storing remote objects.

Functionalities like the generation of stimuli can be translated straightforwardly into HLL functions, as only parameters have to be transferred to the test tool and no evaluation of the results has to be done. Therefore, the corresponding HLL functions just return the special type `TAException` which captures possible communication errors in general. The treatment of checking functions is, however, not that easy. Checking functions usually returns the actual status of a certain property of the system under test, e.g. the content of a GUI text label. The result has to be compared to a reference value to achieve a boolean value that can be mapped to the two possible outgoing branches of a check test block. Therefore the IDL has to be further instrumented, so that for each checking function a corresponding HLL (wrapper) function can be created (see the next paragraph as to how this can be specified). This new function uses the original functionality to retrieve the results. After that it uses a helper function to compare the result to the reference value which was given as an additional parameter, and finally returns a boolean value. All helper functions that are needed by the adapter are declared in a special interface (in Figure 4.20 it is called $\texttt{TestUtil}_1$) which is automatically generated, but needs to be implemented manually. This can be seen as a drawback to this solution. However, with this approach we are able to handle almost every IDL provided by a test tool, and do not have to prescribe test tool vendors how to define their external interfaces.

IDL to Test Block Compiler

For the generation of test blocks from an IDL we have to provide a corresponding test block for each generated adapter function. For the checking functions, see above, we need to provide test blocks, that are also able to specify a reference value for the comparison. Furthermore, we want to allow certain parameters of a test block to be specified as enumerations. This makes the test case design a lot easier afterwards.

All of our instrumentations are defined through special comments in the IDL to ensure that the IDL can be used for other purposes as well, e.g. to compile stubs with the *idl* compiler.

Specifying Enumerations: A certain test block parameter domain can be declared as an enumeration rather than as a string. At the end of the line, where this parameter is declared in the IDL, the following comment has to be added:

/** case: E_1 E_2 ... E_n */

Each E_i represents a single case. Note that it is therefore useful to declare each parameter of a method in a single line within the IDL.

Specifying special check functions: A special check function for an IDL method which compares the result of the original method to (a set of) given reference values can be declared. It extends the signature of the corresponding HLL function with all parameters that are declared within this function. Note that in this case no test block for the original IDL function will be generated.

```
ReturnType f (P1 p1, ..., Pn pn);
/** boolean checkF (P p); */
```

Here the P_i denotes the types of the parameters and p_i the parameter names respectively. Furthermore the return type of `f`, `ReturnType`, and the type of the reference value `P`, declared in the special check method `checkF`, must be compatible in the sense that they can be compared. For the comparison a corresponding function will be defined in the interface:

`<TestToolName>Util`: `bool compute<checkF> (ReturnType c1, P c2)`

The definition of the check function finally results in an HLL function with the following signature:

`%function checkF (P1: p1, ..., Pn: pn, P: p) : (Bool * TAException)`

To support user defined data types in the IDL, the interface `<TestToolName>Util` also creates for each non-primitive data type a function that converts a string into this type. This is because all parameters of test blocks are either of type integer or strings.

Test Block to RTC Compiler

The RTC for test blocks has a recurrent structure:

1. Read out the actual values of the parameters of the test block,

```
beginenv;
    @PARAMETER
    @EXEC
    @BRANCH
endenv;
```

Figure 4.21: Template for the Generation of RTC for Test Blocks

2. call a method of a protocol adapter and pass (some of) the parameters, and

3. check the result to see whether an exception has occurred. If not, evaluate it further to determine the subsequent control flow.

Therefore we propose for the generation of RTC for a test block a template based approach. This enables us to cover almost every standard test tool. There are, however, special test tools that need extra treatment. To support an automatic generation of the RTC for the test blocks for these tools as well, we have developed the *TB-to-RTC* compiler in a general fashion so that it it possible to create specialized instances quite easily.

The most generic template is shown in Figure 4.21. It is comprised only of the portion needed for the execution and evaluation of certain commands. Although it can be used directly for the generation of RTC for almost every standard test tool, templates usually will be enriched by special HLL code that e.g. adds messages to test reports or provides a more sophisticated analysis of exceptions. Every template, used for the automatic generation of RTC, contains up to three special placeholders: `@PARAMETER`, `@EXEC`, and `@BRANCH`. They will be replaced during the generation process with concrete HLL code.

In the first case (`@PARAMETER`) all the values of the parameters of a test block will be stored in local variables. Concretely, for each parameter `p` of a test block the following code will be prepared:

```
var String:  <p> := SD.getSibParameter (Tracer.current_node, "<p>");
```

The keyword `@EXEC` will be replaced with an instruction to execute a command of a protocol. The command is given by the name of the test block (`command`), while the protocol, resp. the name of the adapter, is given by the keyword `Protocol`, prefixed with the class of the test block (`<Class>Protocol`). Note that due to the fully automated generation of the adapters consistency is guaranteed. The return type of the HLL function depends on whether this test block is an action or check test block. In the first case the return type is `TAException`, in the latter a pair consisting of the real result and the exception type `(Bool * TAException)`. The meaning of the parameters was discussed in Section 4.4.3 and it was said that the optional parameters of a test block are finally transferred to the remote test tool, while the others are only used for determining the corresponding test tool. Let $\mathtt{p}_1, \ldots, \mathtt{p}_n$ denote the set of optional parameters, while `toolName` and `commandName` denote "normal" parameters, common to all test blocks[7]. Then the execution code is as follows:

```
rt := <Class>Protocol.<command> (toolName, commandName, <p1>, ..., <pn>);
```

Where the variable `rt` is declared through `var TAException: rt;` or `var (Bool * TAException): rt;` respectively.

Finally `@BRANCH` will be replaced by HLL code to determine the subsequent branch. This again depends on whether this test block is an action or check test block. In the first case `rt` has to be checked to see if a communication problem arises, in which case we proceed with the special error sink `Failure`. If there is no problem we can proceed normally with the branch `default`. When the current test block is a check test block, we additionally have to choose the subsequent branch with respect to the boolean return value of `rt`. The resulting HLL code is then as follows:

```
var Bool: e;
e := ToolAccess.isException (rt);
if (e == false) then
   Tracer.setBranch ("default");
else
   Tracer.setNode (FAILURE);
fi;
```

Action Test Block

```
var Bool: e;
e := ToolAccess.isException (#2.(rt));
if (e == false) then
   if (#1.(rt) == true) then
      Tracer.setBranch ("passed");
   else
      Tracer.setBranch ("failed");
   fi;
else
   Tracer.setNode (FAILURE);
fi;
```

Check Test Block

The class diagram of the generic RTC generator is shown in Figure 4.22. The abstract class `RtcGenerator` provides the generic functionality needed to implement the features discussed above. The RTC can be computed from the template, where the placeholders are replaced with the result of the functions `getParameterHLLCode`, `getBranchHLLCode`, and `getExecHLLCode`. The retrieval of the parameters from a test block is, however, already implemented in the abstract class. The `GenericRTCGenerator` implements the missing functionality in the fashion described above, and can be used for almost every standard test tool. There are, however, special test tools which require more effort for the generation of RTC, cf. Section 4.6. For the implementation of a specialized instance of a RTC generator, a generator simply inherits from either `RtcGenerator` or `GenericRtcGenerator` and is able to overwrite the methods that are responsible for generating the HLL code for the placeholders

[7]Note that all parameter values are stored in local variables, where the names correspond to the test block parameter names.

(`getParameterHLLCode`, `getBranchHLLCode`, and `getExecHLLCode`). Within these methods access to all attributes of a test block is possible, i.e. its name, class, parameters, and its branches.

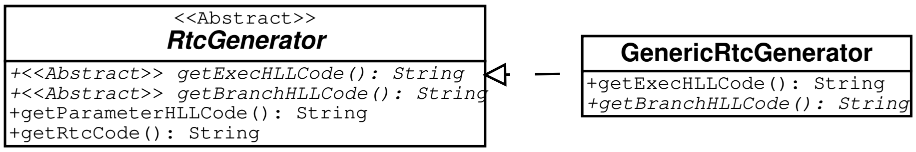


Figure 4.22: *Run-Time-Code* Generator

To sum up, the following steps have to be performed to integrate a new test tool into the *ITE*:

1. Generate the stubs for the IDL's.
2. Compile the IDL's into *ABC* adapter code.
3. Implement a tool facade for accessing the test tool.
4. Implement if required a corresponding utility class for computing the results.
5. Compile the IDL into test block interfaces.
6. Generate RTC for the test blocks.

4.6 Integration of the *Rational Robot* into the Integrated Test Environment

Whereas standard test tools can be integrated into the *ITE* with the tool supported integration process (cf. Chapter 5 and Chapter 6 for examples), within this section we illustrate the integration of the non-standard test tool *Rational Robot* [Rat], capable of the functional regression testing of applications with graphical user interfaces (GUI) running under *Microsoft Windows*[8]. The *Rational Robot* can automatically play back test scripts that emulate user actions interacting with the GUI of the application under test. So in contrast to standard test tools, the *Rational Robot* comprises no fixed set of test actions resp. observation points, but they depend inherent on the concrete application under test.

The *Rational Robot* test scripts can be either programmed in an extension of *Visual Basic* [Mic], called *SQABasic*, or automatically created by capturing user actions.

[8] Another comparable tool is the *Mercury Winrunner* [Mer].

As recorded scripts are also translated into *SQABasic*, they can be changed manually afterwards. Test scripts usually refer to the GUI objects of interest via their (symbolic) component names rather than via their absolute coordinates on the screen. This provides a more general use for the scripts, as they do not depend on e.g. the screen resolution or the applications position on the screen. Before playback, the *Rational Robot* must compile scripts into machine instructions. The validity of the system is determined by comparators at *Verification Points*, where GUI objects (e.g. a certain button or a text field) are compared with a reference of what is expected and usually comprises also non visible properties of such components. Some references are stored in *Verification Files*, while others are embedded in the test script. To sum up the *Rational Robot* is a general test tool which can be used for almost every test scenario where applications with GUI's are being considered.

For the integration of the *Rational Robot* we have to perform the following tasks, cf. Section 4.5:

1. Definition of an IDL for both the test tool and the corresponding protocol(s).
2. Implementation of the CORBA-interface in the *Rational Robot*.
3. Integration of the CORBA-interface into the *ITE*.

4.6.1 Definition of an IDL for the *Rational Robot*

First we have to define an interface to access the test tool itself. In the case of the *Rational Robot* it is not necessary to provide additional functionality that has not already been defined in the base interface `TestToolAccess`. Therefore, it is sufficient to simply inherit from it. For the definition of the supported test protocols we have at least two different possibilities:

1. Define a specific protocol for each considered test scenario, or
2. provide a generic protocol which just supports the transfer and execution of test scripts.

Both possibilities have both advantages and drawbacks. In the first case, we have to define a protocol for every new test scenario that has to be integrated into the *ITE*. Additionally, it is not quite clear where the corresponding test scripts are located. An advantage will be that the protocols can be integrated using the tool supported integration process, presented in Section 4.5.

In contrast, it is also possible to provide a generic protocol only, in principle capable of handling all test scenarios. But then we have to establish a relationship between

```
interface SQAExecProtocol : Protocol {

void execScript (FileBuffer script, SQAParameterList pars, Timestamp execAtTime)
    raises ToolException;

boolean putVerificationFile (FileBuffer verificationFile, string filename)
    raises ToolException;

SQALogMessageSequence getLogMessages (Timestamp startTime, Timestamp stopTime)
    raises ToolException;

};
```

Figure 4.23: IDL for the Test Protocol of the *Rational Robot*

test blocks and test scripts to support an easy and flexible handling of the *Rational Robot*. So in this case we need a compiler that can transform test scripts directly into test blocks. This allows us to extend the *ITE* or more precisely the set of test blocks, quite comfortably. After considering both possibilities we have decided to use this second method. This requires more effort during the integration of the *Rational Robot*, but provides a much more general and flexible usage afterwards. Furthermore, one is not restricted to implementing specialized protocols for certain test scenarios. To allow a broader usage of *Rational Robot* test scripts, we support a parameterization of them. In this way we are able to transfer the (optional) test block parameter at runtime to the test scripts. More technical details about this mechanism are provided in Section 4.6.3.

A shortened version of the general test protocol supported by the *Rational Robot* is given in Figure 4.23. The interface provides methods for the transfer and execution of test scripts, the transfer of verification files, and the retrieval of execution information.

execScript This method allows us to execute a test script within the *Rational Robot*. The parameters of the method are the test script itself, as well as a concrete parameterization for the test script. The parameterization is given by a list of `SQAParameter`, where a `SQAParameter` is a pair of parameter names and their value. Furthermore we can specify a timestamp, that determines the start time of the execution.

putVerificationFile This method transfers a verification file to the *Rational Robot*, which is under certain circumstances needed for the proper execution of a test script. The verification file itself can be given as well as a filename, under which it will be available on the *Rational Robot*. This method is usually invoked before the call of the method `execScript`.

getLogMessages After the successful execution of a test script the results can be retrieved through this method. It can be specified which time slot the results should be gathered from. However in the default case, i.e. empty timestamp, all the results from the last script execution are collected. The result of this method is a sequence of `SQALogMessage`, where each `SQALogMessage` is made up of of an execution status (a boolean value), a message, and a detailed comment.

4.6.2 Implementation of a CORBA-Interface in the *Rational Robot*

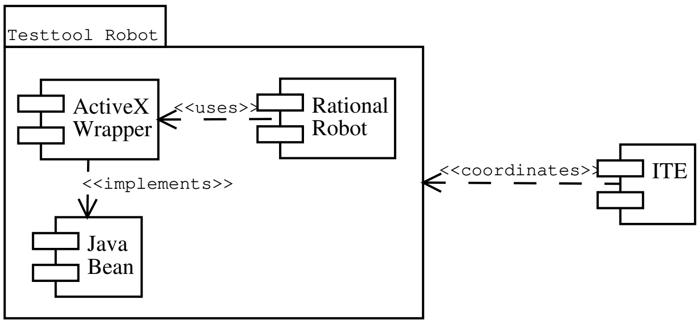


Figure 4.24: CORBA-Implementation in the *Rational Robot*

For the implementation of a CORBA-interface in the *Rational Robot* one has to extend a commercial product designed for testing standalone. To coordinate it from within the *ITE*, a bypass has to be established. Fortunately, the *Rational Robot* supports external *Visual Basic* libraries so that the pattern *extension* can be applied (cf. Figure 4.18). In Figure 4.24 the general architecture of the integration can be seen. The test tool *Robot*[9] consists of the *Rational Robot* itself, together with an external component which implements the CORBA interfaces. This external component is an *ActiveX-Component* [Mic], which is wrapped around a *Java-Bean*. ActiveX-Components and Java-Beans are to some extent comparable component models. In particular, a Java-Bean can be transformed into an ActiveX-Component through the *ActiveX-Bridge* [Sun]. The real implementation of the corresponding IDL is then provided by the Java-Bean. This indirection step is needed as at the moment CORBA is not available for *Visual Basic* directly.

Figure 4.25 illustrates the implementation of the coordination interface within the *Rational Robot*. The *Rational Robot* will be started with a dedicated test script

[9] Note that in the remainder we will often denote the test tool *Robot* with *Rational Robot*, when unambiguous from the context.

(`control`) which initializes the CORBA connection to the *ITE*, i.e. registers the test tool *Rational Robot* at the CORBA nameservice. After that a method of the java bean is called (`waitForScript`), and this brings the *Rational Robot* into an interpreter modus, i.e. it waits until a test script is provided from the *ITE*. Once the *ITE* sends such a test script via the command `execScript`, the *Rational Robot* executes it (`Call script`). Note that the test scripts are transferred at runtime on demand and are in general not physically available to the *Rational Robot* in advance. To stop the *Rational Robot*, a dedicated test script has to be transferred. Finally after a successful execution of the script the *Rational Robot* informs all other components (`scriptExecuted`) and is in the state `waitForScript` again.

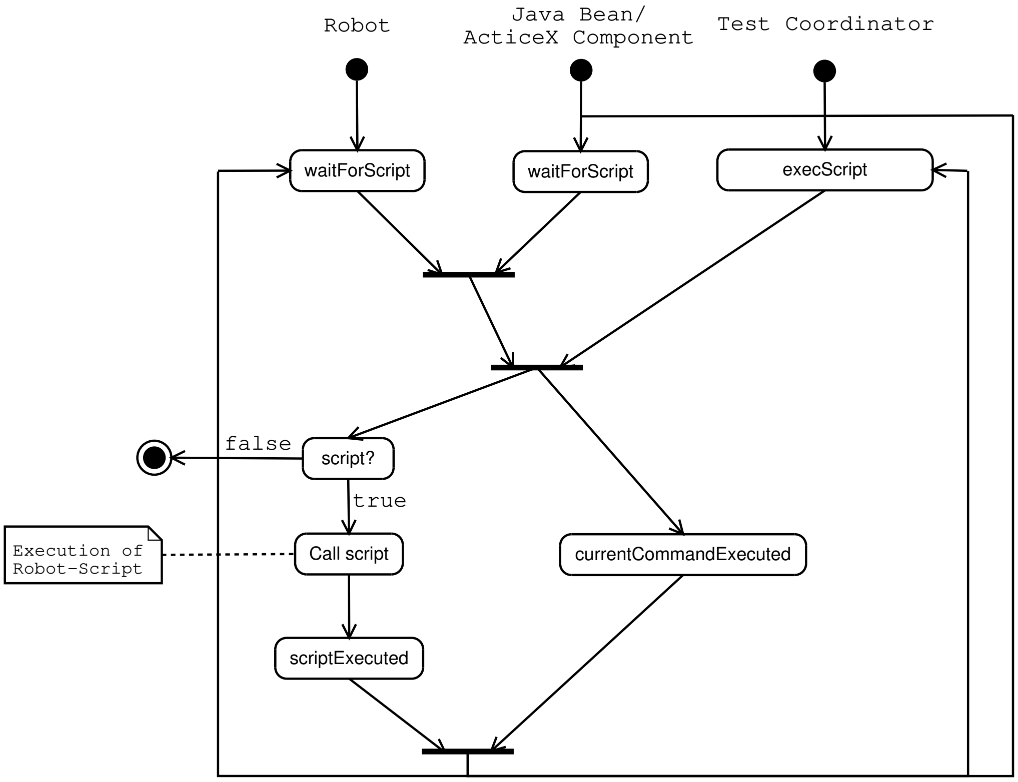


Figure 4.25: Coordination of the *Rational Robot* through the *ITE*

4.6.3 Integration of the CORBA-Interface into the Integrated Test Environment

The tool chain presented in Section 4.5.2 cannot be used for the integration of the *Rational Robot* into the *ITE*. This is because the test functionality is not provided through the IDL but through the test scripts of the *Rational Robot* and this gives more flexibility for extending the test block palette. This way the *Rational Robot* can be used as a generic test tool, capable of testing almost every GUI based application. For the integration there are two tasks:

1. Implementation of adapter code for the IDL `SQAExecProtocol` and
2. generation of test blocks and their RTC for each test script.

Implementation of Adapter Code

Although the general principle of the generation of suitable adapter code for the IDL is similar, the implementation is in the case of the *Rational Robot* more sophisticated, as all three methods of the IDL need special treatment.

execScript When executing a test script the compiled test script has to be transferred to the *Rational Robot* at runtime. It is, however, not suitable to specify the binary code of the compiled test script by a test block parameter. Therefore we propose to specify the (local) filename of the test script only. The HLL function `execScript` is then responsible for loading the binary file and for passing it to the test tool. In addition it is possible to parameterize the test scripts. For this purpose a map of parameters can be passed, realized through two lists of strings, where the first one contains the keys and the second one the values.

putVerificationFile Each test script can refer to several *Verification Files*, where comparators which contain the reference GUI state are stored. The prefix of a test script and the prefixes of its corresponding verification files are identical and these can be used to determine all verification files for a given test script. An additional HLL function `putAllVerfificationFiles` scans the directory where the test script is located for corresponding verification files, loads them, and calls for each the IDL method `putVerificationFile`.

getLogMessages This method allows us to retrieve the results of the execution of a test script from the *Rational Robot*. The original method from the IDL returns all log messages within an interval which is given by parameters. The HLL function, however, is able to retrieve the timing information via the corresponding tool name and the command identifier from the command manager, cf. Figure 4.17. For a further evaluation, an additional HLL type `SQALogMessage` has to be created. The result of the HLL method `getLogMessages` will be a list of `SQALogMessage`.

Generation of Test Blocks for Test Scripts

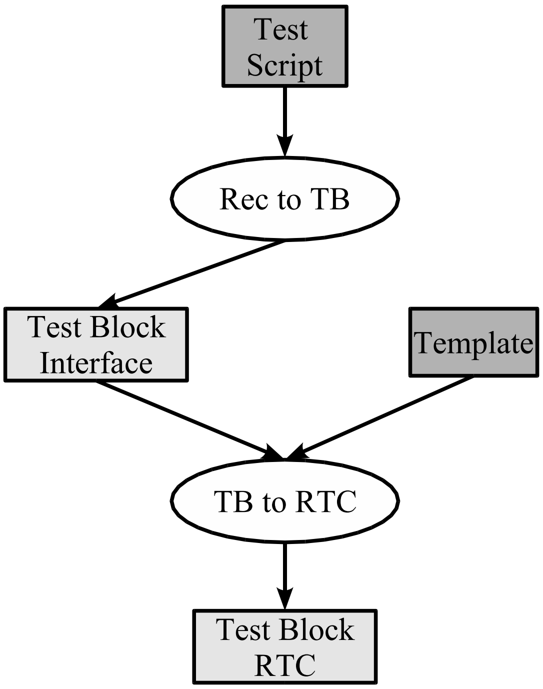


Figure 4.26: Workflow for the Generation of Test Blocks for the *Rational Robot*

To support a convenient usage of the *Rational Robot* by the *ITE*, test scripts have to be compiled automatically into test blocks and their RTC. In Figure 4.26 the general approach is sketched. The test block interface is compiled by the *Rec-to-TB* compiler out of its corresponding test script. The test block interface, together with a special template, serves as a basis for the generation of the RTC through a special instance of the *TB-to-RTC* compiler.

The *Rec-to-TB* compiler is quite simple: it creates a test block for the test script, where the name is just the name of the test script, the class is `SQACommon`, the parameters are `toolName` and `commandName`, and one branch is defined (`default`). This is, however, not always sufficient, in particular one is not able to define check test blocks. Therefore, it is possible to refine each of the elements of a test block interface by means of a special comment at the beginning of a test script[10]. So it is easily possible to define a new name (`#SIB name`) resp. class (`#CLS class`). Furthermore, one can define additional parameters (`#PAR p ...`), which can then be accessed in a test script via the special method `getParam ("p")`. Note that all parameter values are stored as strings. Finally it is also possible to define two

[10]In test scripts for the *Rational Robot* a comment begins with the special character `#`.

outgoing branches instead of the default one. Here it should be noted that test scripts that define more than two branches are rejected by the compiler.

The special *TB-to-RTC* compiler ensures that the specialities of the protocol are taken into account for the generation of RTC. In particular the handling of execution code together with the handling of the branches has to be specialized.

getExecHLLCode The compiler has to determine the full qualified name of the corresponding test script `<name>`, and has to build the parameter map. Let $\mathtt{p}_1, \ldots, \mathtt{p}_n$ be the set of optional parameters of the test block and $\mathtt{v}_1, \ldots, \mathtt{v}_n$ their values respectively. The execution code is then as follows:

```
var TAException:  rt;
rt := SQAExecProtocol.putAllVerificationFiles (toolName, <name>);
if (ToolAccess.isException(rt) == true) then
    Tracer.setNode (FAILURE);
else
    rt := SQAExecProtocol.execScript (toolName, commandName, <name>,
                                       ["<p1>", ..., "<pn>"],
                                       ["<v1>", ..., "<vn>"]);
fi;
```

getBranchHLLCode The HLL code for the determination of the outgoing branches is quite straightforward. First the log messages from the *Rational Robot* are evaluated in the template and the result is stored in a special variable `result`. Then a simple evaluation of the log messages is presented, where the status of the first message determines the overall result:

```
var Bool :  result := true;
var (SQALogMessage List * TAException):  logs;
logs := SQAExecProtocol.getLogMessages (toolName, commandName);
if (ToolAccess.isException(#2.(rt2)) == false) then
    var SQALogMessage:  msg := hd (logs);
    result := SQAExecProtocol.getStatus (msg);
fi;
```

The description of the *Rational Robot* template presented here is rather simple; one may think, however, of more sophisticated evaluation strategies. The HLL code for the branch selection is then as follows:

```
Tracer.setBranch ("default");
```

Action Test Block

```
if (result == true) then
    Tracer.setBranch ("true");
else
    Tracer.setBranch ("false");
fi;
```

Check Test Block

In summary this approach ensures the required flexibility. The integration effort is higher than for standard test tools, but as the *Rational Robot* is a generic test tool, suitable for testing various settings, it is adequate because it only has to be done once. The generation of test blocks directly out of (generalized) test scripts allows an easy integration of new test settings.

Chapter 5

Testing Complex Systems with the Integrated Test Environment

This chapter illustrates the use of the *ITE* by means of a concrete example: the test of a coffee machine server[1]. The example test setting is rather intuitive so that we can concentrate on the tasks that are defined in the test process, cf. Section 3.3.2. First we will introduce the example that will guide us through this chapter in Section 5.1. After that the integration of a particular test tool for this scenario is presented in Section 5.2, using the automated integration process. Finally the last section covers all end-user aspects concerning the use of the *ITE* for the test of complex systems (Section 5.3).

5.1 Introduction

Figure 5.1 shows the architecture of the *Coffee Machine Server* from a logical point of view, i.e. the connections shown denotes no physical connections, as all components are connected through a LAN or WAN. The *Coffee Machine Server* is able to control two instances of concrete coffee machines, i.e. it can switch the coffee machines on or standby, brew coffee, and dispense fresh coffee. The coffee machine server also provides a remote interface via *Remote Method Invocation* (*RMI*) [RMI], so that remote clients can gather the current status of the coffee machines (on/standby, coffee pot full/empty), switch the machine on or on standby, and can brew coffee. It is, however, not possible to dispense fresh coffee via a remote client. The remote access to the coffee machine service for end-user's is provided through a web-based application. This application allows us to connect to the coffee machine server and to

[1]The commonly used vendor machine example in process algebra was the inspiration for this example.

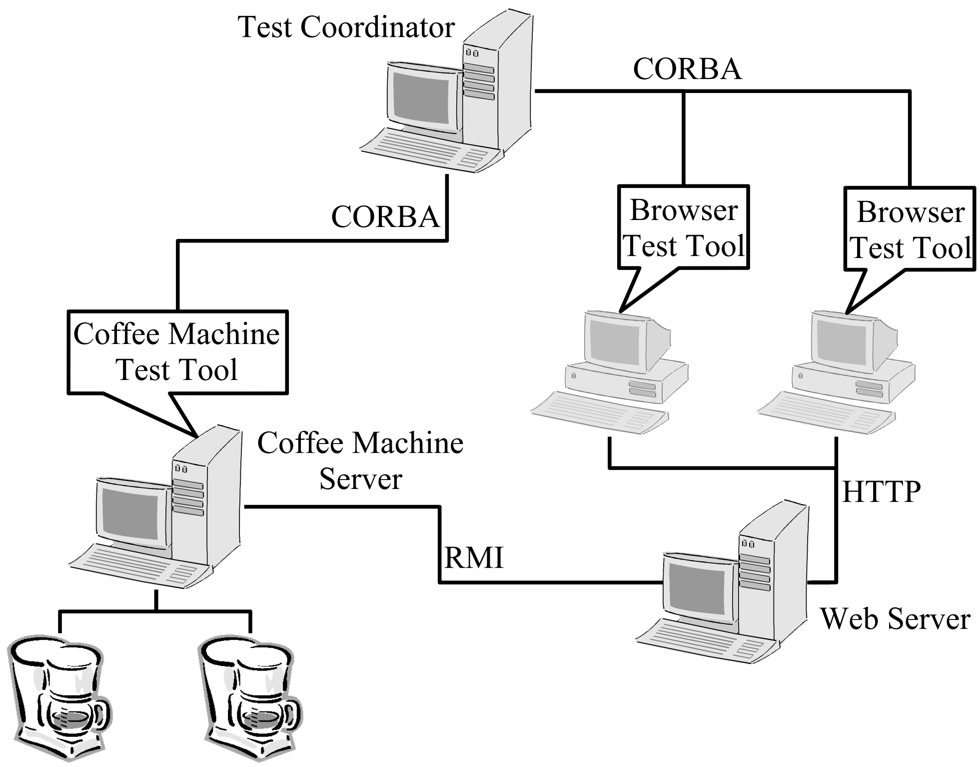


Figure 5.1: Logical Architecture of the *Coffee Machine Server*

brew coffee on one of the corresponding coffee machines. Afterwards it allows us to check whether the coffee machine is free again. To access the web-based application we use two instances of a web browser. Each browser is in the test setting responsible for dealing with a particular instance of one of the coffee machines.

For test purposes the setting described above has to be enriched by several test tools. The *Coffee Machine Test Tool* is responsible for testing the coffee machine server. All possible actions are mapped by the test tool. In addition it provides methods for checking the status of the coffee machines (on/standby resp. full/empty). This functionality will be exported to the *ITE* via CORBA. The web-based application can be tested via a "normal" browser which needs to provide a test interface as well. Within this test interface it must be at least possible to follow links on an HTML page and to check certain properties of them. Again the *Browser Test Tool* can be accessed from the *ITE* via CORBA. Note that it is also possible to use the *Rational Robot* for controlling a browser, as it is a generic GUI test tool.

```
interface BrowserProtocol : Protocol {
    void initBrowser (in string baseURL) raises (ToolException);

    void resetBrowser () raises (ToolException);

    void followLink (in string link) raises (ToolException);

    string getTitle () raises (ToolException);
    /**
        boolean checkTitle (in string title);
    */
};
```

Figure 5.2: Instrumented IDL for the Browser Test Tool

5.2 Integration of the Test Tools

The test process starts with a setup phase, cf. Section 3.3.2. Here it is particularly important to define and implement the test blocks on the basis of the corresponding test tools. With the tool-supported integration process, proposed in Section 4.5, this can be done automatically on the basis of an interface definition of the test tool.

In the next section we will illustrate the integration process for the *Browser Test Tool*, a generic Java browser. This test tool allows us to initialize resp. reset the browser, to follow a link, and to get the title of the currently shown HTML page. In Figure 5.2 the `BrowserProtocol` IDL as a specialization of the generic interface `Protocol` is presented. The methods result in corresponding functions of the protocol adapter, cf. Figure 5.3. For the last method (`getTitle`), an additional adapter function will be created, which returns a pair consisting of a boolean value and the (possible) exception. This function is used for providing the associated check test block, cf. Figure 5.4. It is specified in the comment underneath the original IDL method. In this special case the adapter function `checkTitle` extends the original parameter list of the IDL method with the parameter `title`. Note that although an adapter method for `getTitle` will be created, no corresponding test block will be generated.

```
%function initBrowser (String:  toolId, String:  cmdId, String:  baseURL) : TAException
%function resetBrowser (String:  toolId, String:  cmdId) :  TAException
%function followLink (String:  toolId, String:  cmdId, String:  link) :  TAException
%function getTitle (String:  toolId, String:  cmdId) :  TAException
%function checkTitle (String:  toolId, String:  cmdId, String:  title) :  (Bool * TAException)
```

Figure 5.3: Signatures of the corresponding Adapter Functions

For the computation of the result of the adapter function `checkTitle` a helper method is needed. All helper methods, and generally there are more than one, are gathered in a corresponding utility class called `BrowserUtil`, implemented in C++. The *idl-to-adapter* compiler generates the header files for this class, whereas the implementation body has to be implemented manually. In this special case it results in the following declaration in the header file:

```
static bool computeCheckTitle (const char* cmp_value1, const char* cmp_value2);
```

This method compares two strings and is used in the generated implementation of `checkTitle` to compare the parameter value of `title` with the result of the IDL method `getTitle`.

Furthermore, a local test tool class has to be implemented, cf. Section 4.5.2. This class inherits from a common interface `TestTool`. Basically all relevant functionality has already been implemented in the base class. Therefore we only have to provide special constructors to ensure a proper initialization of the facade class and a comparison operator. So for the test tool `Browser` we need to implement the following methods:

```
BrowserTool (BrowserTool& tool);
BrowserTool (BrowserAccess_ptr access);
void operator= (BrowserTool& tool);
```

Here the class `BrowserAccess_ptr` denotes a pointer to the corresponding protocol `BrowserAccess`, which has been generated from the CORBA *idl* compiler.

Finally in Figure 5.4 the interface of the test block `checkTitle` is shown. The parameter `title` is marked as optional to denote that it is a parameter of the corresponding IDL method. Within the RTC code of the test block the adapter function `BrowserProtocol.checkTitle (toolId, cmdId, title)` is called.

```
SIB checkTitle
CLS Browser
PAR toolName    STR 40 "Browser"
PAR commandName STR 40 "checkTitle"
PAR OPT title   STR 40 ""
BR passed
BR failed
```

Figure 5.4: Interface of Test Block `checkTitle`

5.3 Using the Integrated Test Environment

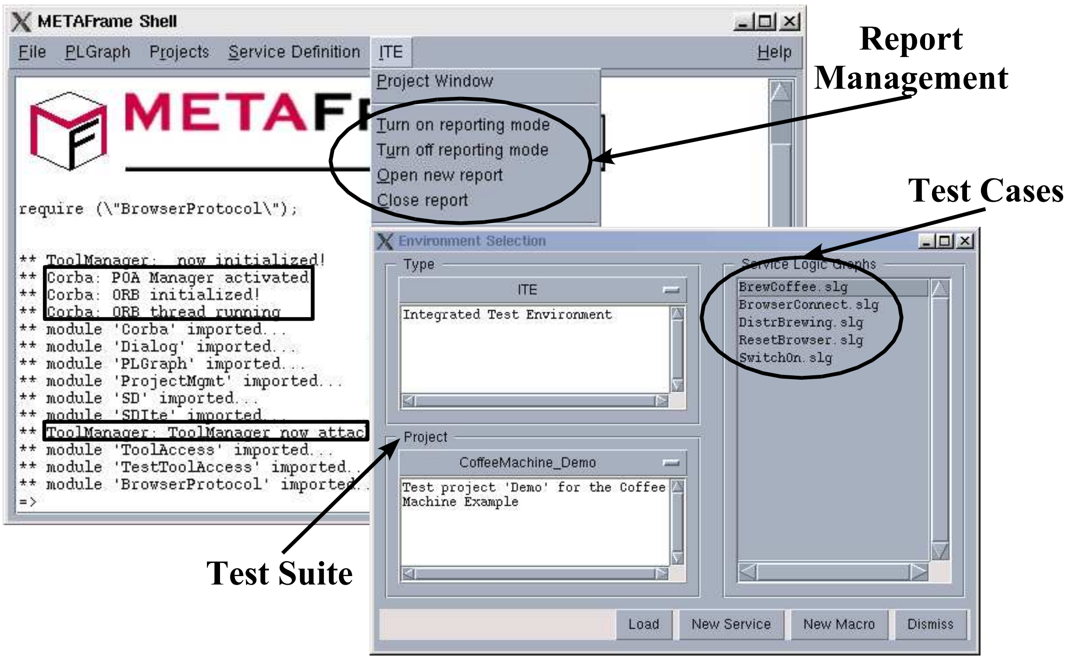


Figure 5.5: Main Window of the Test Coordinator

The main window of the test coordinator can be seen in Figure 5.5. It consists of the *ABC* interpreter window in the background and a project window. In the *ABC* interpreter window several information messages are logged. In particular the currently loaded modules are presented to the user, together with additional diagnostic information. In this special case it is noted that the CORBA initialization was successful and that the `ToolManager` is now attached to the CORBA nameservice. In addition, the *ITE* instance of the *ABC* interpreter provides a special menu `ITE`, which covers all global aspects of the *ITE*. The menu allows users to access the project window which manages test cases. Within the *ITE* test cases are organized by test projects, where a single test project is in some respects comparable to a test suite, as they both combine test cases. But test projects offer more functionality than plain test suites, as they also organize test blocks and constraints. Furthermore test projects are hierarchical, meaning that all defined test blocks and constraints of a test project can be included together. In principle it is possible to include test cases as well, but they are suppressed in the project window to keep it manageable. The test project window allows us to browse all available test projects, to create new test cases, to load existing ones, and to create macros which can later be used as atomic test blocks.

By the `ITE` menu of the *ABC* interpreter it is also possible to control the report module. The user can turn the report mode on or off, which is useful when developing new test cases. Furthermore, a user can open new reports or can close active ones. The report management can also be controlled via specific test blocks in the test cases directly.

5.3.1 Defining Properties using the Constraint Editor

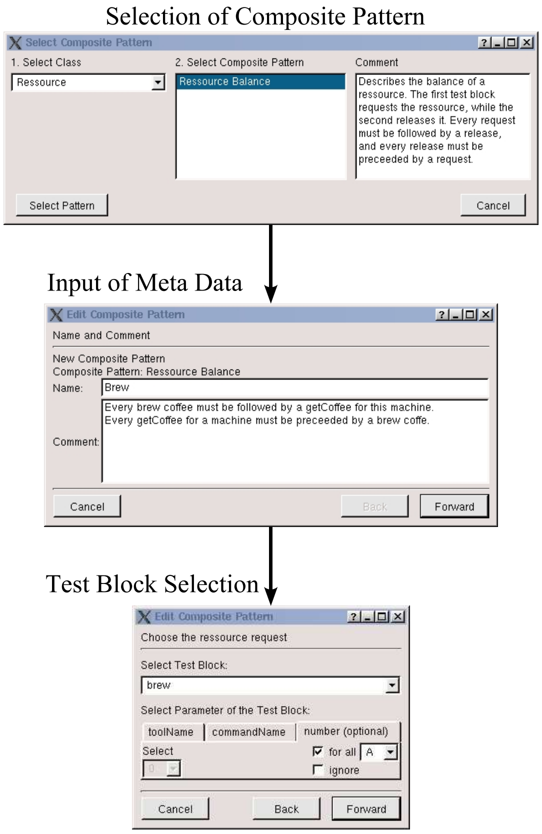


Figure 5.6: The Constraint Editor of the *ITE*

After the definition of the test blocks, the test process continues with the definition of constraints. For this purpose the *ITE* provides a special editor that offers a graphical user interface on top of the pattern system based on XML documents (see Section 4.4.2). A few dialogs of the constraint editor are shown in Figure 5.6. The definition of a new constraint starts with the selection of an appropriate composite pattern. As discussed in Section 4.4.2, composite patterns are organized via classes. After the class has been chosen, a pattern of this class can be selected. A detailed description is presented in the rightmost text field, to help a user during the selection process. Once a composite pattern has been chosen, the user is asked to provide some meta data, i.e. a *name* and an informal *description* has to be given. Finally the user must fill the slots of the pattern with concrete test blocks. At the bottom of Figure 5.6 a concrete one is presented. Here the test block that allocates or requests the resource has to be selected for the composite pattern *Resource Balance*, cf. Example 3.4. In Figure 5.6 the test block `brew` is chosen. The definition of this pattern prescribes the dialog `SelectTBWithParameters`, i.e. the user has to specify a test block together with its parameters (cf. Figure 4.15). All parameters of the test block `brew` are listed at the bottom of this dialog. For a single parameter either a concrete value can be given (`select`) or the check box `for all` denotes that it will be bound to the quantifier. The generated constraints will then be immediately available within the *ITE*.

5.3.2 Designing Test Cases

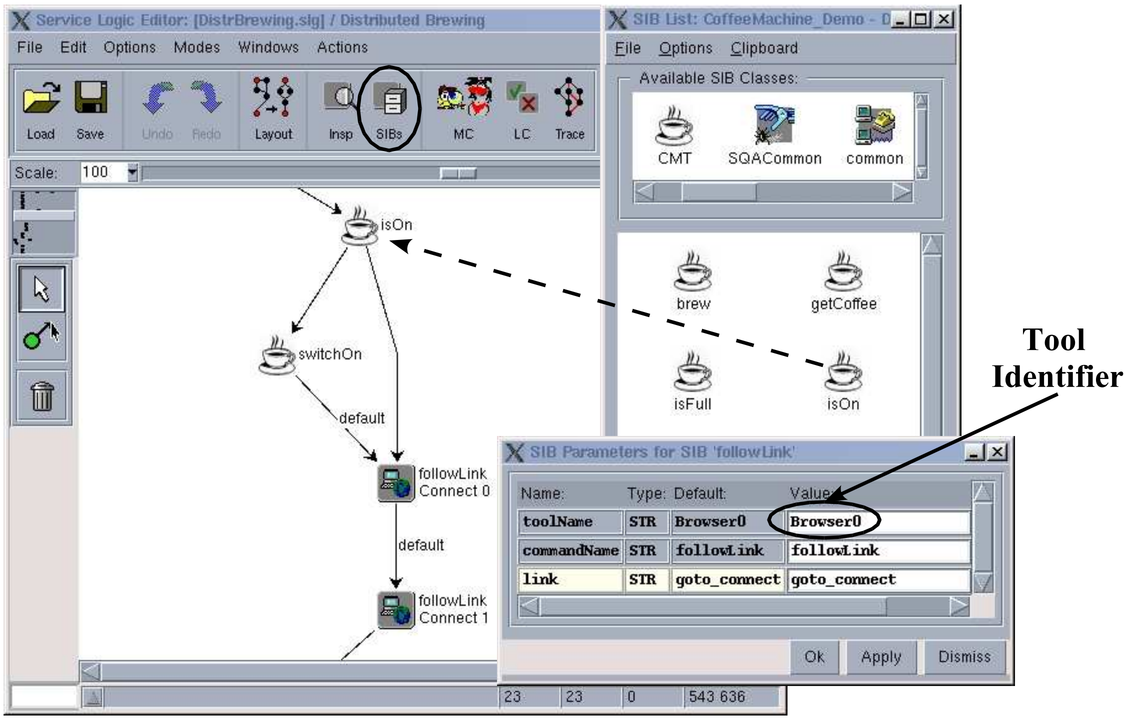


Figure 5.7: Designing Test Cases

As shown in Figure 5.7 the test blocks occurring in test cases are presented to the user according to their classes, in palettes accessible from the test coordinator's GUI. Note that different test block classes are represented through different icons. Users construct test cases by drag-and-drop of the desired test blocks from a palette onto the canvas. The control flow design to steer the execution is also done graphically, by defining

— the workflows contained in the test case. This is done by connecting the test blocks through edges, and

— the data and events steering the branching, done by configuring each test block's internal parameters.

The test block parameter can be configured via a special dialog presented in Figure 5.7. It allows us to set all parameters for the chosen test block. Here a tool identifier has to be specified to identify the corresponding test tool at execution time. In the example presented here the first `followLink` test block will be executed on the tool with the identifier `browser0`, whereas the second one at `browser1`. To support user readable test cases, the test blocks in a test case can be renamed to e.g. give a hint as to which tool instance will execute the action or which link will be followed.

Furthermore the editor provides functionality, common to each graphical editor:

— Loading and saving of test cases,

— an undo/redo mechanism, and

— a highly configurable layouter module.

For a detailed discussion of this features please refer to [MET99].

A concrete test case is shown in Figure 5.8. Every test case starts with an initialization phase, where all used test tools are initialized. The first test block (`startTestCase`) is an internal one which mainly declares the two special exception states `failure` and `commFailure`. These two error sinks allow a sophisticated error diagnosis. While the first one captures problems that occur within a test tool resp. the system under test, the second one catches general communication failures, i.e. CORBA exceptions. The initialization phase proceeds with the initialization of the test tools needed in the test case. Here two test tools are prepared through `prepareCMT` and `prepareBT`. A `prepare<Tool>` test block binds a test tool to a symbolic name through the `ToolManager`. Afterwards the test tool `Browser` needs to be initialized before being used.

After the initialization phase, in this example a single stimulus is sent to the system, i.e. the coffee machine will be switched on through the browser. During the evaluation phase, the responses will be checked by the coffee machine test tool, i.e. whether the coffee machine is now properly switched on or not. Note that even this

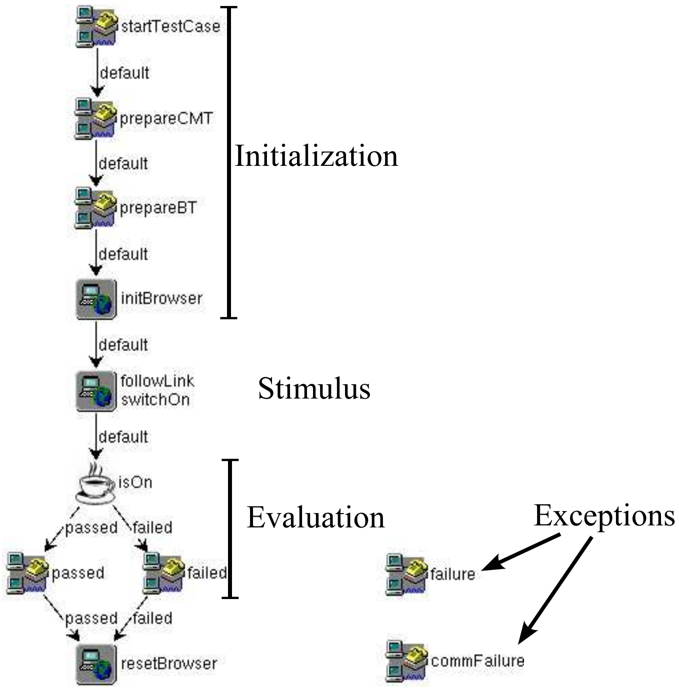


Figure 5.8: Example of a simple Test Case

simple test case captures the essence of complex systems, i.e. the stimulation by one test interface (browser) and the observation by another (coffee machine). Finally a reset will be performed on the `Browser`.

5.3.3 Verifying Test Cases

After their design the test cases are subject to verification. Here both local and global properties of the test cases are checked. Both checks can be initiated directly from within the editor through the `LC` resp. `MC` button. Figure 5.9 shows the results of a local check, where here concretely two errors were detected. The first one concerns the parameterization of the marked test block `brew`, where the parameter `toolName` is unset. This check for a correct parameterization is the most common application of local checking. The other error, however, concerns the start node, which is not marked as such. The corresponding local check code assures that every test block without ingoing edges has to be marked as a start node. Note, however, that the second erroneous node is not visible within the screenshot of Figure 5.9. When checking local properties for each error the corresponding node is usually shown in the editor.

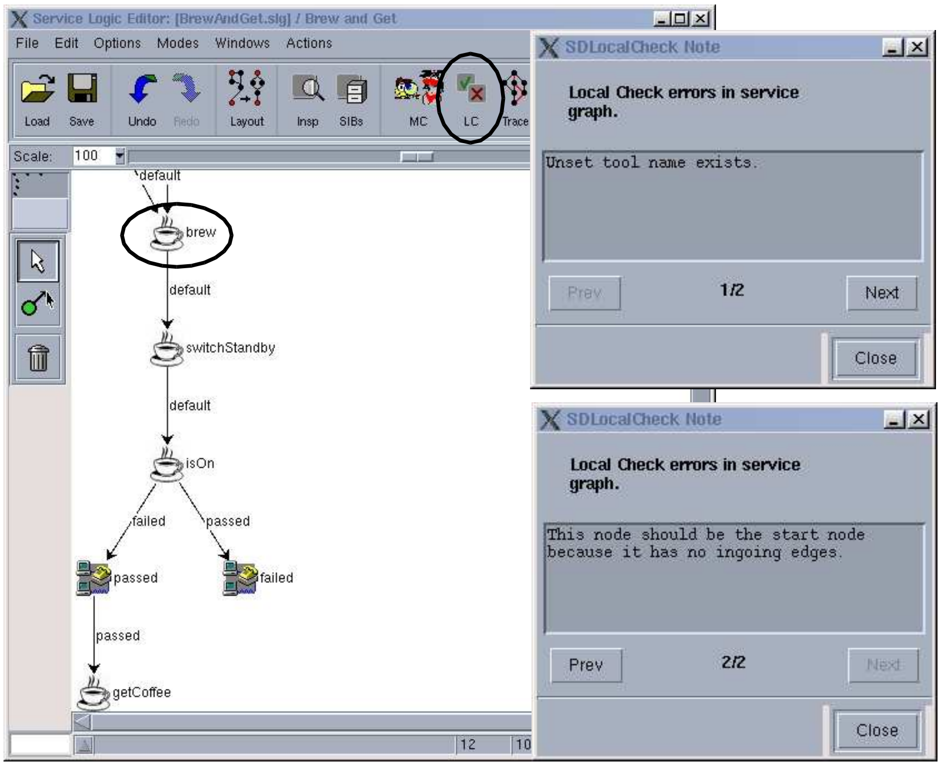


Figure 5.9: Local Check of a Test Case

After the correction of the errors, that were found during the local check, the global check will be initiated. The constraints are organized in a constraint database, see Figure 5.10. Constraints are also part of a test project, so that only the constraints of the current active test project and all included ones are visible. The user chooses one or more constraints of interest out of the constraint database, expressing single *aspects* of the test case under construction, and checks their correctness online. Note that the test engineer is not bothered with the concrete syntax of the constraints, but only with their abstract description. If a test case violates a constraint detailed diagnostic information concerning the mistake and its possible location will be presented, cf. Figure 5.10. This is repeated until all relevant aspects have been treated. Due to the online verification with the model checker, constraint violations are immediately detected at design time.

The failure detected in Figure 5.10 is a good example of commonly detected errors. The user takes the coffee out of the machine in the `passed` path, but has forgotten to release the coffee pot after the test case has been evaluated to `failed`. This, however, usually affects the execution of further test cases, as the overall system has not returned to its initial state.

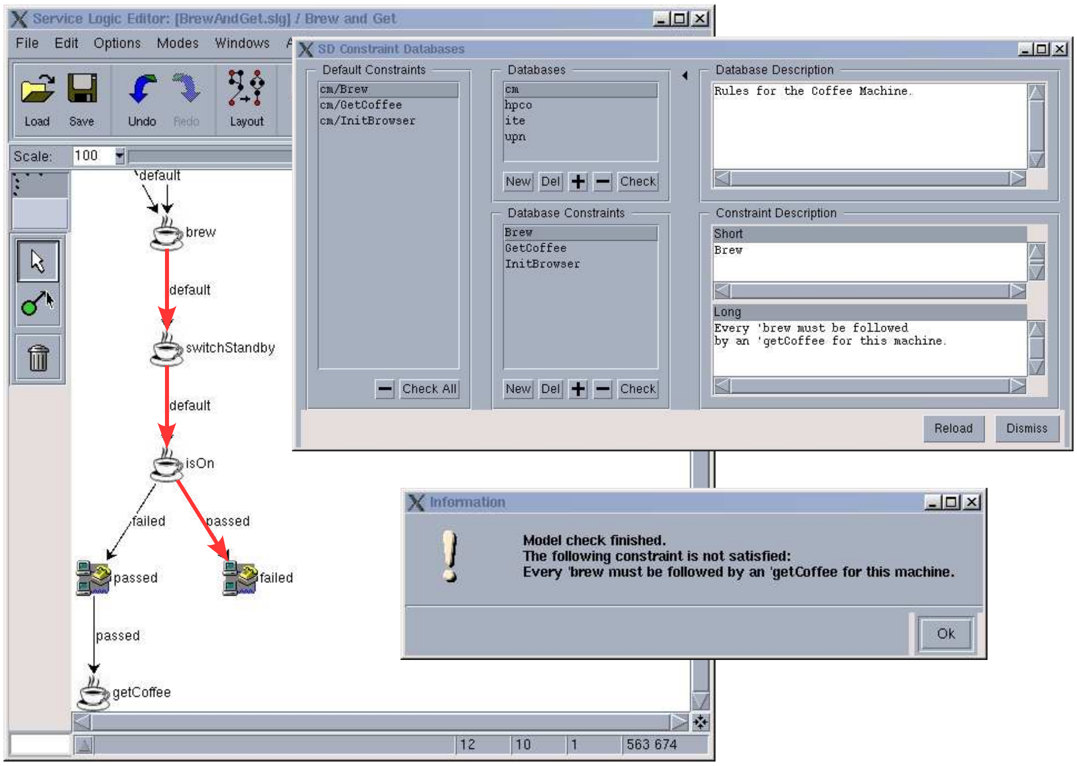


Figure 5.10: Global Check of a Test Case

5.3.4 Executing Test Cases

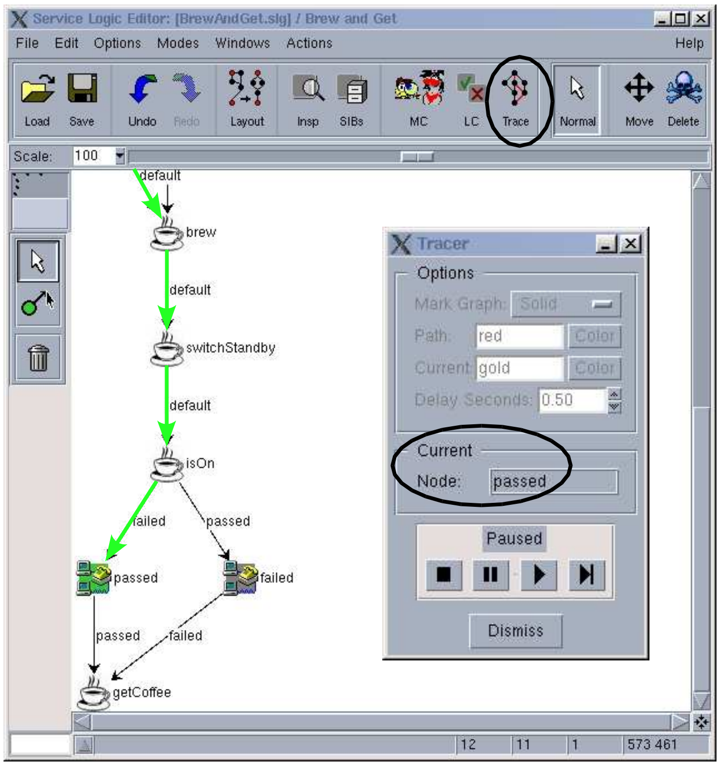


Figure 5.11: Executing Test Cases

Once the test cases have passed the verification step they can be immediately executed by means of the `Tracer`. Starting at a dedicated test block, the tracer proceeds from test block to test block. The current execution path is marked in the test case. The execution can be controlled through the `Tracer` dialog, cf. Figure 5.11. Here detailed parameters about the execution can be set, e.g. the delay between each test block execution. Furthermore, the execution can be done step by step to have full control, or automatically. The test block being carried out is marked in the test case and is also presented in the middle of the dialog.

5.3.5 Analyzing Test Cases

The *ITE* provides a web-based application for the management of test reports. It is designed as a role-based, proactive web service, which supports the cooperation process of teams of test engineers. We focus on the role-based and permission-based management of the test protocols: the web service provides several roles with different rights to access test reports and functionalities. This ensures that all users

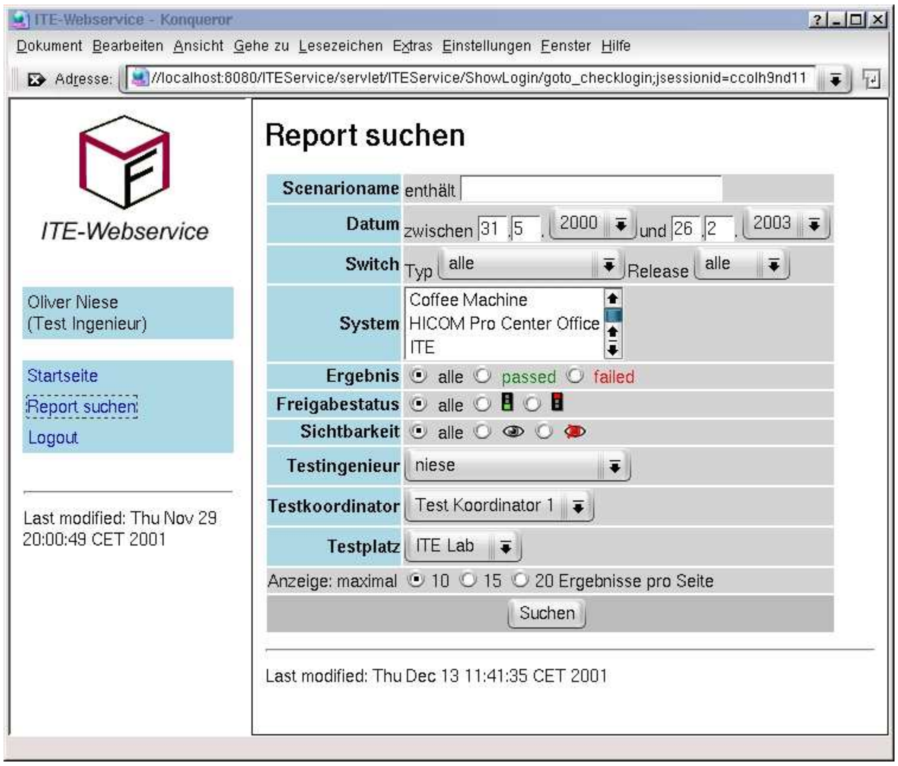


Figure 5.12: Web-based Application for the Management of Test Reports

of the service get a tailored view of the scenario of their interest, which guarantees data security and easy handling: only the functionalities (e.g. show test report, search, delete, and modify) are presented to a user and these are supported by the current role. Figure 5.12 presents a screenshot of the application. It provides a comfortable search mask for accessing the database, e.g. concerning the date of test runs, the test result, the system under test, etc. Test reports can be modified so that different attributes can be set (e.g. the status) and comments can be added. Due to the XML based storage of the test protocols the *ITE* web service is able to present different views of the same underlying sources: the service provides different XSL ([Worb]) style sheets that generate different HTML outputs for the presentation of the data. E.g. Figure 5.13(a) shows a test report at the detailed level required for a test engineer, whereas Figure 5.13(b) shows a guest user's view, as provided by another style sheet. It only presents a rough overview of the test suite.

ITE-Testreport

26.02.2003

Übersicht

System-under-test: CoffeeMachine_Demo

Übersicht

System-under-test:	CoffeMachine Demo
Scenario:	BrewAndGet
Testdurchführung:	26.02.2003, 16:44:53
Testplatz:	niese@fiege.cs.uni-dortmund.de

Ausführungsergebnis:

Anzahl Testgraphen (inkl.Init.):	1
Erfolgreich ausgeführte Testfälle:	1
Fehlgeschlagene Testfälle:	0

Testfallergebnisübersicht:

Ausführungszeitraum	Testgraph	Testgraphkommentar	Ergebnis
26.02.2003 16:44:54 - 26.02.2003 16:44:57	BrewAndGet_correct.slg	Brew and Get	Passed

Testprotokolle

Testprotokoll: BrewAndGet_correct.slg

26.02.2003 16:44:54	**BrewAndGet_correct.slg** (Brew and Get)
26.02.2003 16:44:54	**isOn** *toolName*: CMT, *traceMark*: , *commandName*: isOn, verfolgter Kontrollflusszweig: *passed*
26.02.2003 16:44:54	**brew** *toolName*: CMT, *traceMark*: , *commandName*: brew, *number*: 0, *tracetime*: 0, verfolgter Kontrollflusszweig: *default*
26.02.2003 16:44:54	**isFull** *toolName*: CMT, *traceMark*: , *commandName*: isFull, *number*: 0,

(a) Detailed Test Report

ITE-Testreport

26.02.2003

Übersicht

System-under-test: CoffeeMachine_Demo

Übersicht

System-under-test:	CoffeeMachine Demo
Scenario:	Test Suite BrewAndGet
Testdurchführung:	26.02.2003, 16:45:44
Testplatz:	niese@fiege.cs.uni-dortmund.de

Ausführungsergebnis:

Anzahl Testgraphen (inkl.Init.):	6
Erfolgreich ausgeführte Testfälle:	4
Fehlgeschlagene Testfälle:	2

Testfallergebnisübersicht:

Ausführungszeitraum	Testgraph	Testgraphkommentar	Ergebnis
26.02.2003 16:45:45 - 26.02.2003 16:45:52	BrewAndGet_correct.slg	Brew and Get	Passed
26.02.2003 16:45:52 - 26.02.2003 16:46:02	BrewCoffee.slg	Untitled	Passed
26.02.2003 16:46:02 - 26.02.2003 16:46:18	DistrBrewing.slg	Untitled	Passed
26.02.2003 16:46:18 - 26.02.2003 16:46:58	BrewAndGet.slg	Brew and Get	Failed
26.02.2003 16:46:58 - 26.02.2003 16:47:10	BrewAndGet.slg	Brew and Get	Failed
26.02.2003 16:47:10 - 26.02.2003 16:47:27	DistrBrewing.slg	Untitled	Passed

(b) Compressed Test Report

Figure 5.13: Test Reports

Part III

The Integrated Test Approach – Practice

Chapter 6

Testing Computer Telephony Integration Solutions

Testing *Complex Telephony Integration* solutions is a multidimensional task which demands automation via adequate system-level tool support. The complexity lies in the interaction between the components, i.e. a classical telephony switch in cooperation with applications, as well as in the short innovation cycles and the great number of possible combinations between the telephone switch and the applications. In previously published articles we have discussed the use of the *ITE* along industrial applications, which we have done in cooperation with *Siemens AG* [Sie]. There we have focussed on the test of a complex call center solution [NMH+01, HMN+01] and on the test of a virtual switch architecture together with a web-based call management application [MNSE02, MNS02b]. The results presented in this chapter, however, go beyond that, as we will discuss the general concepts of testing *Complex Telephony Integration* solutions in more detail and present additional practical results.

This chapter starts with a description of the application domain that we are considering: *Computer Telephony Integration* solutions (Section 6.1). After that in Section 6.2 we discuss the specialities when testing such systems and present a suitable test architecture. In Section 6.3, 6.4, and 6.5 we present results from case studies we have done together with *Siemens AG*. The chapter concludes with an evaluation of the use of the *ITE* for the test of *Computer Telephony Integration* solutions, where we discuss aspects concerning the design of test cases, as well as economic aspects (Section 6.6).

6.1 Computer Telephony Integration Solutions

The world of telecommunications has rapidly evolved during the last 15 years, modifying in this process its focus. In 1985 a telephone switch was “only” used as a telephone switch. Additional components, either hardware or software, were gradually developed to bring additional functionality and flexibility to the traditional switch, e.g. in the initial days voice mail or billing systems. Today not only single functionalities are added at a quick pace, but the switch is mutating its role into the central element of complex heterogeneous and multivendor systems: it is nowadays integrated into whole business solutions, e.g. in the field of hotel solutions, call center and unified messaging applications. These solutions are called *Computer Telephony Integration* (*CTI*), i.e. systems consisting of telephony hardware in cooperation with software applications, often called *value-added applications*. CTI solutions aim to support the workflows involved in the use of telephones by means of applications, e.g.

— Support of the establishment of telephone connections through an application where e.g. a number can be dialed from within an electronic address book.

— Automatic dialing, e.g. in a call center, an application organizes the process of connecting free call center agents to customers from a call list.

— Incoming calls will be announced, and additional information concerning the conversational partner will be presented within an application, where the partner is identified by his or her call number.

— Intelligent call forwarding, based on e.g. the incoming call number or on an automatic interaction with the caller.

— Logging of communication details, e.g. the call parties, call duration, further notes, etc.

The first attempt to implement CTI solutions was to directly connect computers to a telephone switch or *Private Automatic Branch Exchange* (*PABX*), to create an added value for the PABX, cf. Figure 6.1 (left). Here the interaction between the PABX and the applications, located on a dedicated application server, was almost exclusively implemented via proprietary interfaces. Today the definition of open standards like e.g. the *Computer Supported Telecommunication Applications Protocol* (*CSTA*) [Eur94, Eur98], which specifies command sets and the data structures needed for telephony applications, or *Telephony Application Programming Interface* (*TAPI*) [Mic01], which defines a programming interface for telephony applications, pushes the field towards the development of new, system-level applications. Nowadays a PABX is just another node in an IP based network, and the communication

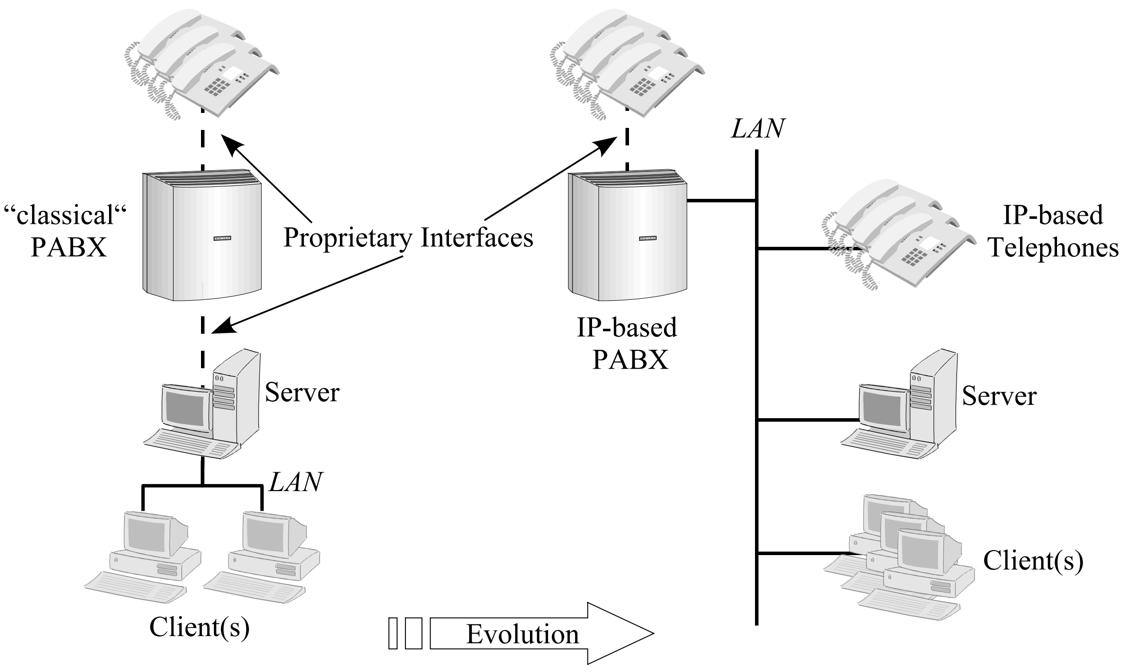


Figure 6.1: Evolution of CTI Platforms

between the PABX and its (IP-) telephones and application servers resp. clients takes place on the LAN, as depicted in Figure 6.1 (right).

The diagram in Figure 6.2 (left) documents the trend towards a growing product integration in terms of the increase in the number of value-added applications that work in average on or with a PABX. As one can see, the integration factor has been rapidly increasing since 1995, and the trend points in the direction of even larger value-added product ranges.

A parallel but concurring aspect is the increasing number of major PABX releases per year (cf. Figure 6.2 right). This trend is driven mainly by the convergence between classical telecommunication technology and modern IP technology, e.g. "Voice-over-IP": a modern PABX is itself a complete complex system, and experiences the accelerating evolutionary pace of hardware and software in combination! In order to summarize the overall complexity of CTI solutions, we must look at the interaction between the components as well as in short innovation cycles and the great number of possible combinations between the PABX and the value-added applications.

A typical example of a CTI setting is illustrated in Figure 6.3, showing a midrange PABX and its environment. The PABX is connected to the *Integrated Services Digital Network* (*ISDN*) or, more generally, to the *Public Switched Telephone Network*

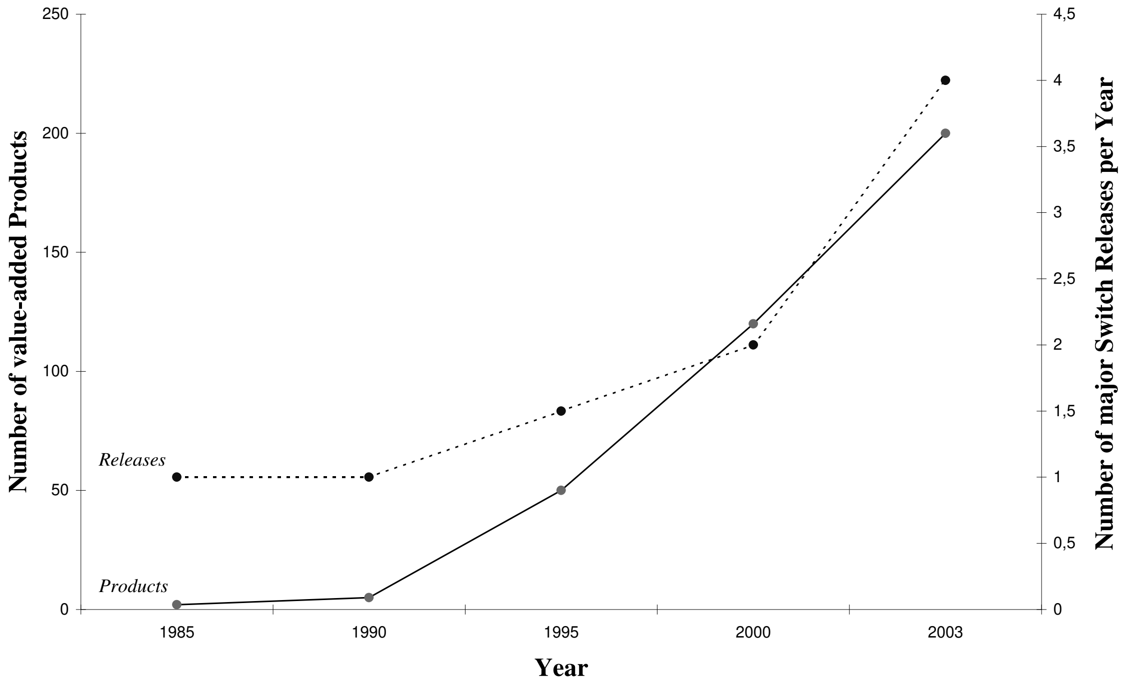


Figure 6.2: Trends in the development of CTI systems: (left) growing product integration and (right) faster paced PABX releases

(*PSTN*), and acts as a "normal" telephone switch to the phones. In addition, it communicates directly via a LAN or indirectly via an application server with CTI applications that are executed on PCs. Like the phones, CTI applications are active components: they may stimulate the PABX (e.g. initiate calls), and they also react to stimuli sent by the PABX (e.g. announce incoming calls). Therefore in a system-level test scenario it is necessary to investigate the interaction between such subsystems.

Even the relatively simple scenario of Figure 6.3 demonstrates the complexity of CTI platforms from the communication point of view, because there are several (internal) protocols involved. E.g. the telephones communicate via the *Corporate Network Protocol* with the PABX, whereas the PABX communicates via *CSTA Phase II/III* protocol with the application server. On the application server, a *TAPI service provider* performs a mapping of the CSTA protocol to the TAPI protocol, which is the communication protocol between the application server and its clients.

In the rest of this chapter we first briefly recall the *Open Systems Interconnection* basic reference model for communication protocols, before we focus on some technical details concerning the involved protocols.

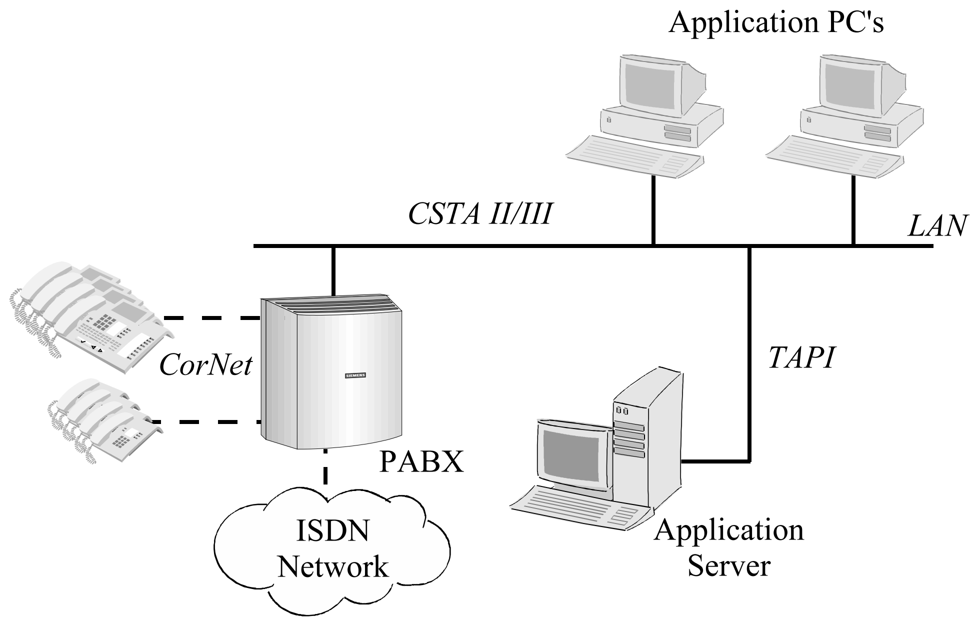


Figure 6.3: Example of a CTI Setting

Open Systems Interconnection – ISO/OSI Basic Reference Model

The objective of the *Open Systems Interconnection* (*OSI*) [ISO84] model is to provide a set of design standards for equipment manufacturers, so they can communicate with each other. The OSI model defines a hierarchical architecture that logically partitions the functions required to support system-to-system communication. The OSI model has seven layers, each of which has a different level of abstraction and performs a well-defined function, cf. Figure 6.4. The principles that were applied to arrive at the seven layers are as follows: a layer should be created where a different level of abstraction is needed; each layer should perform a well-defined function; the layer boundaries should be chosen to minimize the information flow across the interfaces; and finally the number of layers should be large enough that distinct functions do not have to be thrown together in the same layer, and small enough that the architecture does not become unwieldy.

The layered approach offers several advantages. By separating networking functions into logical smaller pieces, network problems can be more easily solved through a divide-and-conquer methodology. OSI layers also allow extensibility. New protocols and other network services are generally easier to add to a layered architecture.

The application, presentation and session layers comprise the upper layers of the OSI Model. Software in these layers performs application-specific functions like data formatting, encryption, and connection management. The transport, network, data

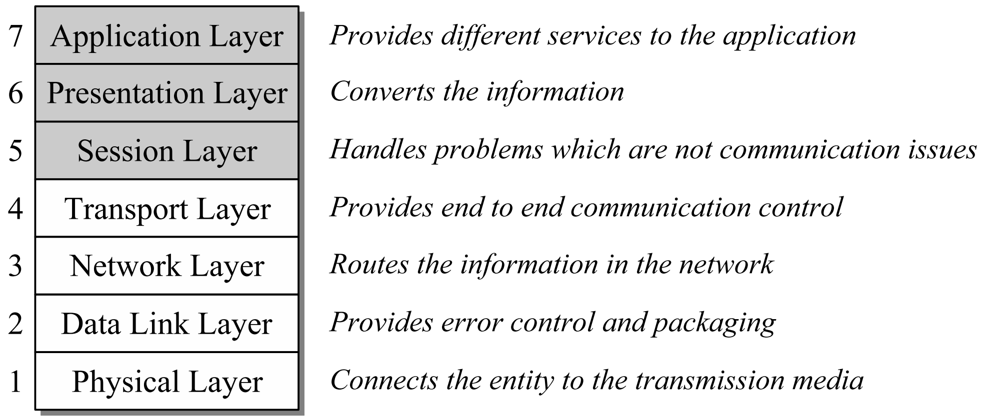


Figure 6.4: Open Systems Interconnection – ISO/OSI Basic Reference Model

link, and physical layers comprise the lower layers, which provide more primitive network-specific functions like routing, addressing, and flow controls.

Integrated Services Digital Network Protocol

The *Integrated Services Digital Network Protocol* (*ISDN*) [CCI89a, CCI89b] is a design for a completely digital telephone/telecommunications network. It is capable of carrying voice, data, images, video, etc. It is also designed to provide a single interface (in terms of both hardware and communication protocols) for hooking up a phone, a fax machine, or a computer. ISDN is based on a number of fundamental building-blocks. First, there are two types of ISDN "channels" or communication paths:

B-channel The Bearer ("B") channel is a 64 kbps channel which can be used for voice, video, data, or multimedia calls. B-channels can be joined together for even higher bandwidth applications.

D-channel The Delta ("D") channel can be either a 16 kbps or a 64 kbps channel used primarily for communications (or "signaling") between switching equipment in the ISDN network and the local ISDN equipment. In Europe the *DSS-1* protocol [CCI93] is used for the corresponding control protocol.

These ISDN channels are delivered to the user in one of two pre-defined configurations: *Basic Rate Interface* resp. *Primary Rate Interface*. The difference between these two configurations lies in the number of available B-channels (2 resp. 30 B-channels). The physical interface of ISDN is called *S0* resp. *S2M*.

It must be noted that in contrast to the reference model of [ISO84] for ISDN the D-channel is only defined up to layer 3 and the B-channel only for layer 1. So the ISDN protocol is responsible for the management of connections, whereas the layers 4 to 7 are only implemented in e.g. the telephones.

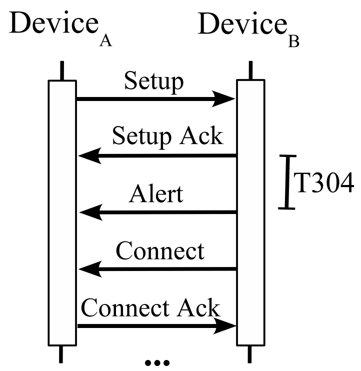


Figure 6.5: Establishment of a connection with DSS-1

The establishment of a connection between two ISDN devices according to the DSS-1 protocol is depicted in Figure 6.5. A time-sensitive interplay of protocol messages is needed for this purpose. Note that we present the connection procedure in a somewhat simplified manner, as in the real implementation more timers are used. First the initiating device ($\texttt{Device}_A$) sends a `Setup` command. Now the called device ($\texttt{Device}_B$) reacts to the `Setup` with a `Setup Acknowledge`. After the `Setup Acknowledge` the called device has to send an `Alert` to the initiator within a well-defined amount of time (specified by means of the timer `T304`). After the initiator receives this message, it can present it to the user, e.g. by changing the ring tone. Finally after a `Connect` from the called device, i.e. the user has answered the incoming call, a `Connect Acknowledge` establishes the connection.

Corporate Network Protocol

The *Corporate Network Protocol* (*CorNet*) is a proprietary ISDN oriented D-channel layer 3 protocol, defining the communication between a *Siemens* PABX (*Hicom 150e*) [Sie] and its telephones. CorNet can be transported on the physical resp. link layer through ISDN, as well as through TCP/IP or even RS.232. The CorNet protocol basically offers commands for handling resp. controlling a telephone, e.g.

— commands to lift up the receiver or replace it, type digits, and where applicable alphanumeric symbols or special buttons,

— signals for modifying display lines, brightness and colour of lamps, or ringer mode and ringer pattern,

— commands for the activation resp. deactivation of specific features of the PABX.

Computer Supported Telecommunication Applications Protocol

The *Computer Supported Telecommunication Applications Protocol* (*CSTA*) [Eur94, Eur98] specifies an application interface and protocols for monitoring and controlling calls and devices in a communications network. These calls and devices may support various media and can reside in various network environments such as IP, Switched Circuit Networks and mobile networks. CSTA, however, abstracts various details of underlying signalling protocols (e.g. SIP/H.323) and networks for the applications. CSTA is an application layer protocol and the CSTA protocol messages can be transported on the lower layers again through ISDN, TCP/IP, and RS.232.

Depending on how one looks at it, CSTA defines a telephony-process model for applications and a computing process model for the PABX. In our case, we are interested in CSTA-services as well as in the CSTA-protocol: by means of the protocol an application accesses the CSTA-telephony-services from the PABX or provides CSTA-computing-services to the PABX. Each model consists of a set of objects and rules to change the states of the objects. Examples of telephony objects include

Device Objects representing anything that allows users to access telecommunications services. They can be either physical (buttons, lines, and stations) or logical (a group of devices, e.g. an automatic call distribution (ACD) group). A device has attributes, including *device type*, *device identifier*, and *device state* that can be monitored and manipulated by an application.

Call Objects describing logical sessions among calling and called parties. The call behaviour, i.e. its establishment and release, can be observed and manipulated by an application. A call object representing a call session has attributes such as *identifier* and *state* and offers operations such as *make call* or *clear connection*. One or more devices may involved in a call in different phases.

Connection Objects representing a relationship between a call and a device. It is characterized by attributes such as *identifier* and *state* and by operations such as *hold* or *clear*. Many CSTA-services, such as hold-call, reconnect-call, or clear-call, operate directly on connections.

Basically, CSTA-services consist of a request and a response, both possibly parameterized. A monitor service can be activated to track control and other activities and to receive notification of all changes in the PABX. Starting a monitor indicates that an application, be it a component of the system or an external observer, wants to be notified of changes that occur in calls, devices or applications, and device attributes managed by the PABX. Examples of changes include arrival of a call at a device, answering a call, and changing a device by modifying features such as “forwarding”. Event reports are sent to the monitor-requestor.

Telephony Application Programming Interface

The *Telephony Application Programming Interface* (*TAPI*) [Mic01] provides a uniform set of commands for any supported telephony device that is connected to a computer. TAPI bridges the gap between the (abstract) telephony hardware related protocol CSTA to a concrete programming interface for building applications through a *TAPI Service Provider*.

TAPI is designed to establish connections between end-points on a telephone network and provides programming access to three objects for call control: *addresses*, *lines* and *calls*. End-points are called *addresses*, which can be represented by a phone number. It is possible to have more than one address on a single *line*, where a line is the physical connection between each end-point and the PABX. So one location can have multiple lines and one line can have multiple addresses. Furthermore, one address can have multiple *calls* and it is possible to switch between calls, by placing one on hold before activating another. With this technique one can also place a call on hold, establish another call and transfer the initial call to the new address, or join two or more calls together and have a conference. These and many other advanced call control techniques are supported by TAPI.

6.2 System-level Testing of Complex Computer Telephony Integration Systems

In the rapidly evolving scenario discussed in Section 6.1, the need for efficient automated regression testing is evident: whenever a release occurs, either of the PABX or of (a subset of) the application programs that cooperate with it, – singularly or, increasingly, in collaborative combinations – the correct functioning of the new configurations must be certified again.

The following properties are of particular interest when testing CTI solutions:

Interdependencies The test of interdependencies between the actions of a test case or even between actions of different test cases is needed to be able to con-

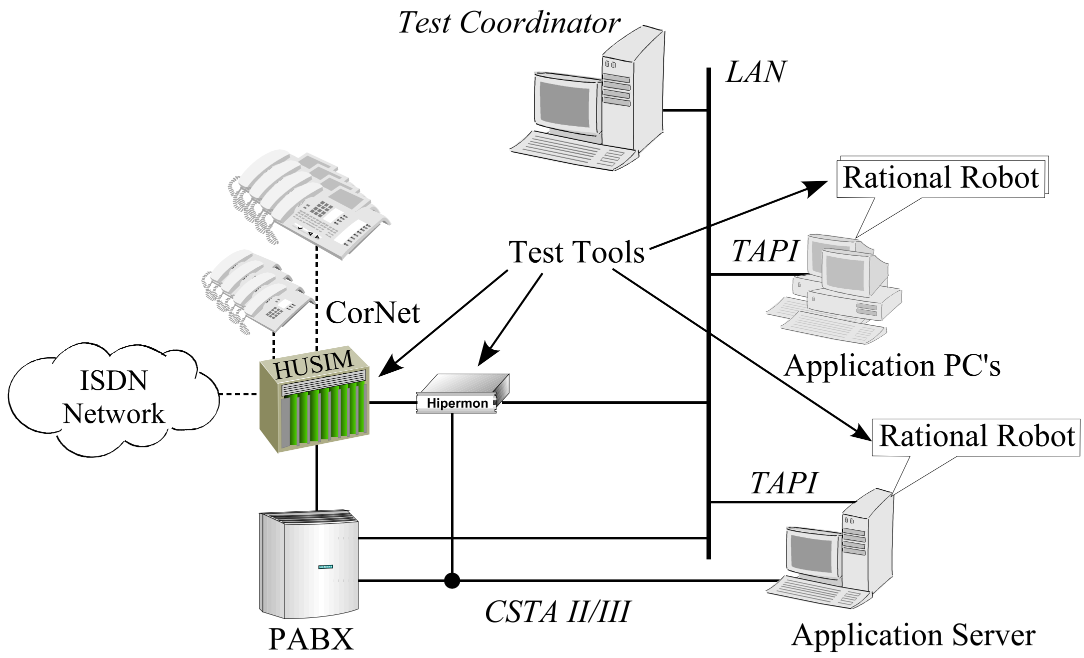


Figure 6.6: Concrete Test Setting for a CTI Solution

tain the feature interaction problem, which occurs usually in "non-standard" situations.

Assumptions The test of assumptions about the state of the system's resources is required, since this is distributed as well. Within this class of testing properties it must be ensured that the assumptions about the systems state are correct, i.e. the states of the applications and the PABX are synchronized.

Admissibility Criteria The test of admissibility criteria for single operations is important because of the widely distributed character of the solutions.

To test the properties discussed above the complex interplay between protocols, on different levels of abstraction, must be considered, and it is clearly unfeasible to do this at the level of customary, finely grained protocol analysis. This is why the *ITE* is particularly well suited to deal with CTI systems. Within the *ITE* it is possible to cover the overall system in a uniform way, i.e. test blocks for e.g. the DSS-1 protocol (ISDN, level 3) can be used in conjunction with application level test blocks, e.g. which stimulate the application directly[1].

[1] Usually stimulating a CTI application results in corresponding CSTA protocol messages (level 7).

Additional scale complexity is introduced by the test tools themselves: for each significant test interface a specific, dedicated test tool participates in the regression test. Note that some test tools are able to control/observe more than one interface. Concretely, the simple scenario shown in Figure 6.3 grows in the test laboratory to the dimensions depicted in Figure 6.6, whereby each of the devices and applications must be set up, steered, and reset during system-level regression test.

In the scenario considered here three different kind of test tools are supported by the *ITE*:

Hicom Environment Simulator

The *Hicom Environment Simulator* (*HUSIM*[2]) is a simulation device used for quality assurance in small telecommunication systems. The scope of HUSIM is to perform tests on a PABX of type *Hicom 150e*, where it simulates the behaviour of a variety of devices:

— analog and digital telephones,

— analog and digital trunks,

— proprietary digital telephones,

— proprietary DECT base station and handsets,

— printers, account devices, CSTA devices, etc.

The HUSIM can perform operations such as: user off hook and on hook, selections, trunk calls, speech tests, DECT base station calls, ISDN calls and so on. The commands for the HUSIM are gathered in test scripts which can be sent to the HUSIM by means of a special command line test tool. During test execution, the messages sent from the PABX are received by the HUSIM and sent back to the test tool. The test tool can decode the messages and e.g. store them in a plain text file. The test evaluation will be done by means of a comparison of the result file with a previously recorded reference file. This low level of abstraction during test case evaluation is the main reason why we do not propose to integrate the HUSIM directly. Instead, the HUSIM will be controlled by another test tool (Hipermon), which establishes a level of abstraction higher than that of the HUSIM.

[2]In German it denotes **H**icom **U**mwelt **Sim**ulator.

Hicom Protocol Emulator and Real-time Monitor

The *Hicom Protocol Emulator and Real-time Monitor* (*Hipermon*) [Her] is a proprietary test tool, well suited for the testing of CTI systems. It is capable of recording (filtered) traces of protocols on different system interfaces simultaneously. In particular the Hipermon is able to trace the CorNet, DSS-1 and CSTA protocols on the system interfaces LAN, S0/S2M, and V.24 (serial interface). So in the sense of an *ITE* test tool the Hipermon supports three test protocols, each on three different system interfaces.

The test functionality of the Hipermon is based on the HUSIM. The Hipermon, however, provides a higher level of abstraction for the level 3 protocols than the HUSIM does.

CorNet For the CorNet protocol, the Hipermon establishes a state-oriented view on top of the message-oriented protocol. So the Hipermon keeps a record of all sent and received protocol messages and is able to determine the actual state of all involved telephones, given through their individual connection state, display content, and LED state. With this abstraction a telephone can be fully controlled and observed from the end-user's perspective, i.e. it is possible to hook off resp. hook on, to dial a number, to press all available buttons, and also to check the current state of the display (-lines), to check the state of the LED's and the ring-tone.

DSS-1 For the DSS-1 protocol, commands for the management of connections are offered by the Hipermon. Again it is important to provide the commands from the end-user's perspective, as the detailed, time sensitive interplay of protocol messages needed for the establishment of a connection between two devices (cf. Figure 6.5), is not suitable to be used within *ITE* test cases. Instead the Hipermon provides two special commands for this purpose, i.e. `makeCall` and `acceptCall`. The command `makeCall` implicitly activates a separate task that waits for a `Connect` and sends the corresponding `Connect Acknowledge` within the protocol conform timeouts. Furthermore, the command `acceptCall` sends first a `Setup Acknowledge`, and after a well-defined period of time an `Alert` followed by a `Connect`. Note that it is also possible to answer an ISDN call to an internal telephone with the corresponding CorNet commands. Additionally the Hipermon provides commands for clearing resp. rejecting connections, and is able to report on the current call-state of a device.

CSTA The commands for the CSTA protocol can be mapped directly by the Hipermon, as it is already an application layer protocol. It is, however, necessary to provide check functionality that allows an abstraction of too detailed information, e.g. concrete timestamps or connection identifier.

The Hipermon also provides functionality to e.g. manage the underlying trace buffer (`reset`, `retrieve`, and `save`) and to obtain the corresponding test protocols (`get-CorNetProtocol`, `getDSS1Protocol`, `getCSTAProtocol`, and `getHUSIMControl-Protocol`). The latter provides commands to activate the HUSIM trace explicitly. This is needed for certain HUSIM variants that are unable to trace all the time. In this case, before a stimulation of the system through an application takes place, the HUSIM must be armed in order to trace the corresponding responses of the PABX.

Overall the Hipermon provides 31 different test blocks for stimulating and observing telephone devices (without CSTA). Please note that the integration of the Hipermon exactly follows the automated integration process, defined in Section 4.5.

Rational Robot

The *Rational Robot*, as a general purpose GUI test tool (cf. Section 4.6), will be used in several instances to steer and observe the involved applications, located either on the application server or on a client.

In addition the test coordinator has access to the PABX itself, e.g. to perform an initialization at the beginning of a test case execution.

6.3 Testing the HiPath ProCenter Office

The *HiPath ProCenter Office* (HPCO) of *Siemens AG* [Sie] is a total solution for call center and messaging applications for small and medium size companies. It offers wide-ranging application possibilities of individually matching all communication processes to corporate workflows and optimizing them. HPCO offers both conventional call center functions such as *Automatic Call Distribution* (*ACD*) with routing to groups of experts and messaging applications such as *VoiceMail*, *eMail*, *FaxMail*, and *SMS* (*Short Message Service* via GSM).

6.3.1 Introduction to HPCO

HPCO consists of a PABX of type *Hicom 150e* and several applications running on either an application server or a client. The client applications are used in several instances, one for each involved call center agent. Additionally every call center agent is equipped with a telephone. For the test of the HPCO four different applications are considered: *ACD Supervisor* (application server), *Tray Phone*, *ACD Agent*, and *Communications* (all client PC, cf. Figure 6.7).

ACD Supervisor This application allows the supervisor or administrator to configure agents, groups, etc., and to actively influence the communication process in the call center. Furthermore, it provides detailed information about the call center in forms of reports and statistics. During the testing of HPCO this application is used to initialize the call center, so that it is in a well-defined configuration.

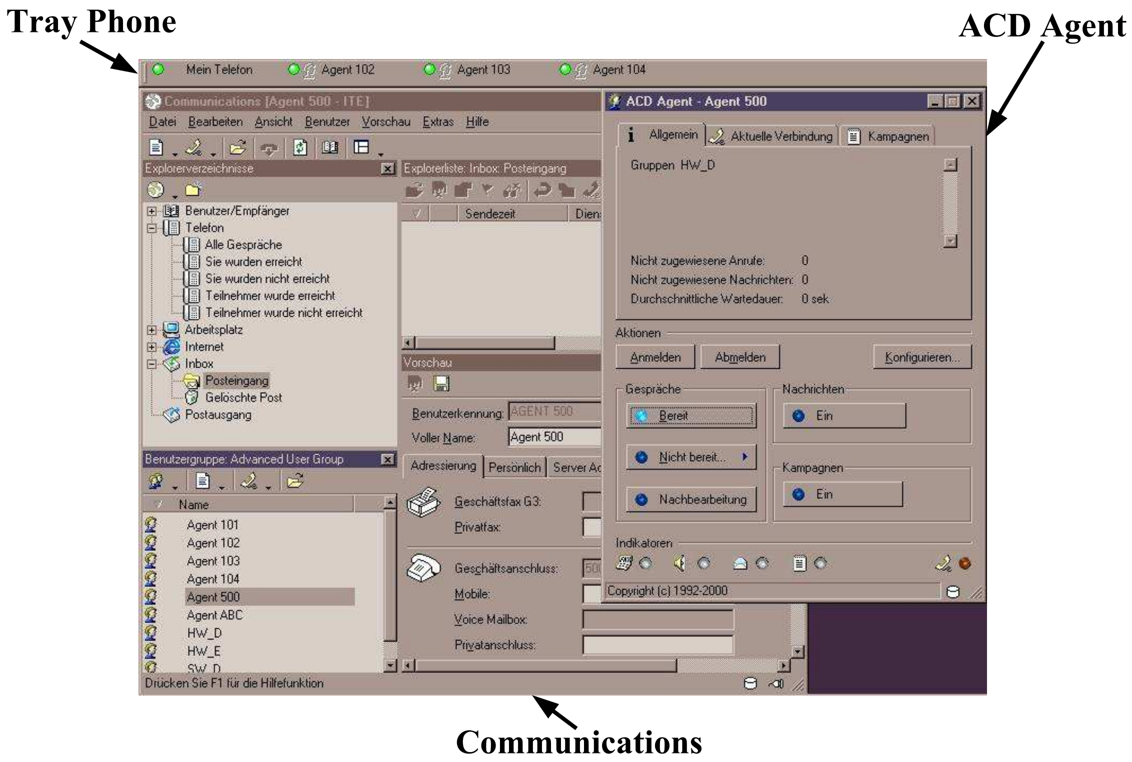


Figure 6.7: Client Applications of the HPCO

Tray Phone The *Tray Phone* application manages calls, i.e. initiating new calls, holding calls, transfering calls to other agents, or initiating conference calls. Furthermore, it provides status information about other agents in the call center, e.g. whether the agent is active, off duty, or in a call.

ACD Agent This application allows a call center agent to login to resp. logout of the call center easily, to set his status to active or off duty, etc. Additionally it provides information about the duration of active calls by means of a progress bar and the duration of the post processing of calls.

Communications The *Communications* application offers sophisticated messaging functions, like *VoiceMail*, *FaxMail*, *eMail*, and *SMS*. It also offers statistical information about the groups the call-center agent is a member of, and about the agent itself.

Furthermore, the telephone of a call-center agent continuously shows information relevant to the agent, i.e. group association, current call status, waiting calls, and his own status.

6.3.2 Test Setting for the HPCO

When testing HPCO an instance of the test setting of Figure 6.6 can be directly used, where the application *ACD Supervisor* resides on the server, and the client applications are distributed among the clients.

Relevant testing properties concerning the client applications are, e.g.:

— The *Tray Phone* application signals the actual status of several call center agents, e.g. concerning the call state. Therefore, the interdependency between the *Tray Phone* and the physical phones must be tested. This can be done e.g. by initiating a call via the telephones and validating whether this call is visualized correctly by the application, or by performing a call by means of an interaction between a telephone and *Tray Phone*.

— The state of the application and the state of the PABX resp. of a single telephone needs to be synchronized. For this purpose an application activates a monitor service and uses the resulting CSTA event-reports to update its status. Missing CSTA events, however, can lead to inconsistent states and therefore to typical error situations that have to identified throughout the testing phase.

— In the HPCO, groups of experts can be established. In the test setting it must be ensured that only members of such a group can accept calls intended for this group. In other words a call to an expert group must not be sent to a member of another group.

6.3.3 Evaluation

For the test of HPCO 171 different test cases were designed. Additionally 95 new test blocks for the test of the involved applications were developed and they have been implemented, without exception, with the help of the *Rational Robot*, i.e. test scripts for the *Rational Robot* have been recorded and were automatically compiled into test blocks. Overall 2371 instances of the test blocks have been used for the implementation of the test cases, which implies that a test case is composed of an average of approximately 14 test blocks. The distribution of the used test blocks, i.e. how many test blocks are PABX-related, for environmental purposes, or application specific, is depicted in Figure 6.8. Over 57% of test blocks are reused, i.e. not implemented for this test setting only. The high degree of environmental, i.e. internal, test blocks is remarkable. Considering the concrete number of test blocks it turns out, that in an average test case (14 test blocks) 5 test blocks are internal ones. These test blocks, however, are necessary in every test case, i.e. `StartTestCase` for the test case initialization, `commFailure` and `failure` as exceptions, and `passed` and `failed` for test case evaluation. Furthermore, as seen in Figure 6.8 the test cases focus on the behaviour of the CTI applications, as twice as many test blocks concerning the applications are used than PABX related ones. Note that the behaviour of the CTI applications, however, depends on the interaction with the PABX.

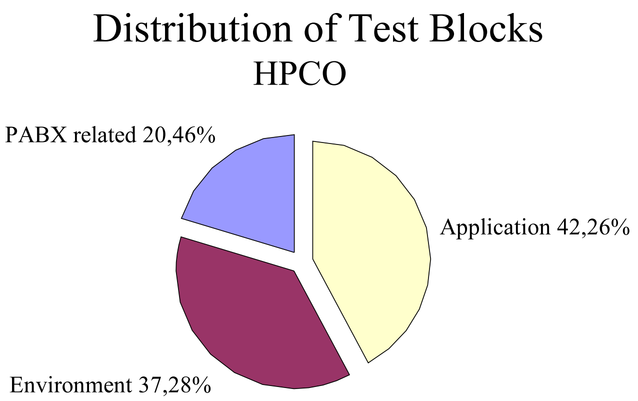


Figure 6.8: Distribution of used Test Blocks: HPCO

6.4 Testing the Personal Call Manager

An example for the convergence of classical telecommunication systems with web-based applications is the *Personal Call Manager* application [Sie01]. It allows users to comfortably reconfigure their settings at any time and independent of their actual location, as they can access this application via the internet with a normal web browser.

6.4.1 Introduction to the *Personal Call Manager*

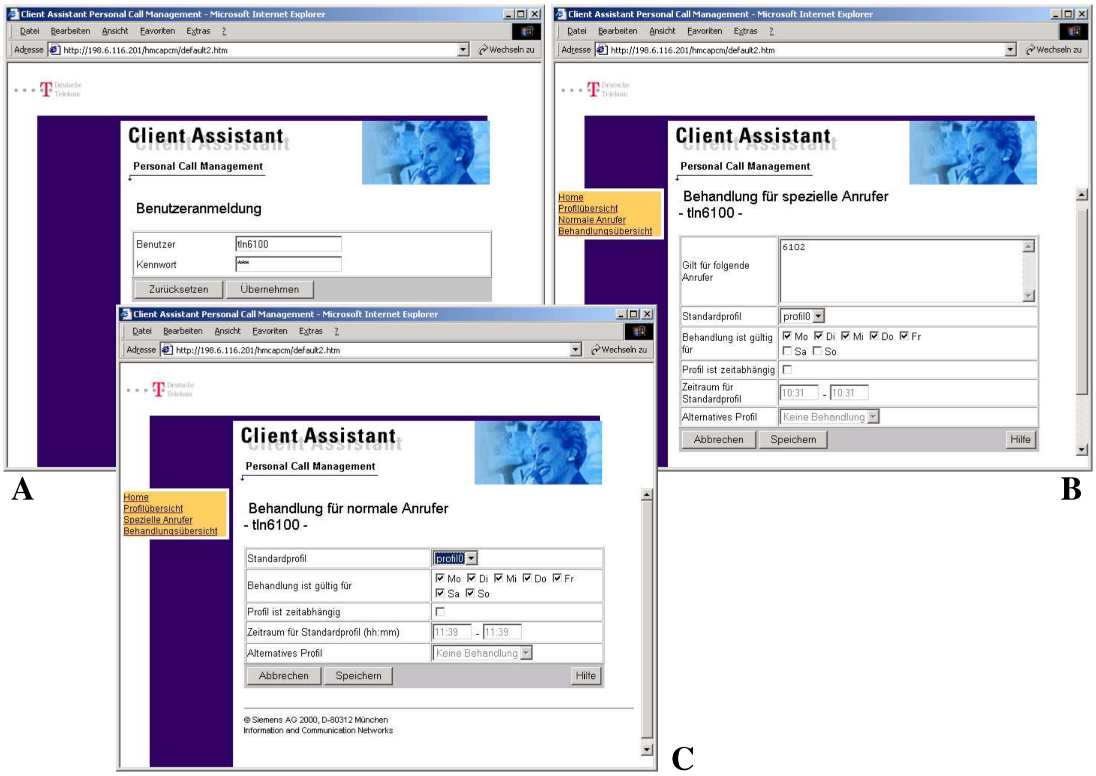


Figure 6.9: The Personal Call Manager

The *Personal Call Manager* application is particularly useful in enabling users to manage a *qualified* call-forwarding: this takes into account characteristics like the identity of the caller or the current time. It can forward the call with respect to those conditions to different numbers, it can handle user-defined profiles connected with specific forwarding conditions (forwarding policies) as well as a well-organized user-defined exception handling of those policies (cf. Figure 6.9(B,C)). Additionally, it supports a user management that allows protecting the access with personal passwords, and a role management that respects the association of specific rights to different user groups (cf. Figure 6.9(A)).

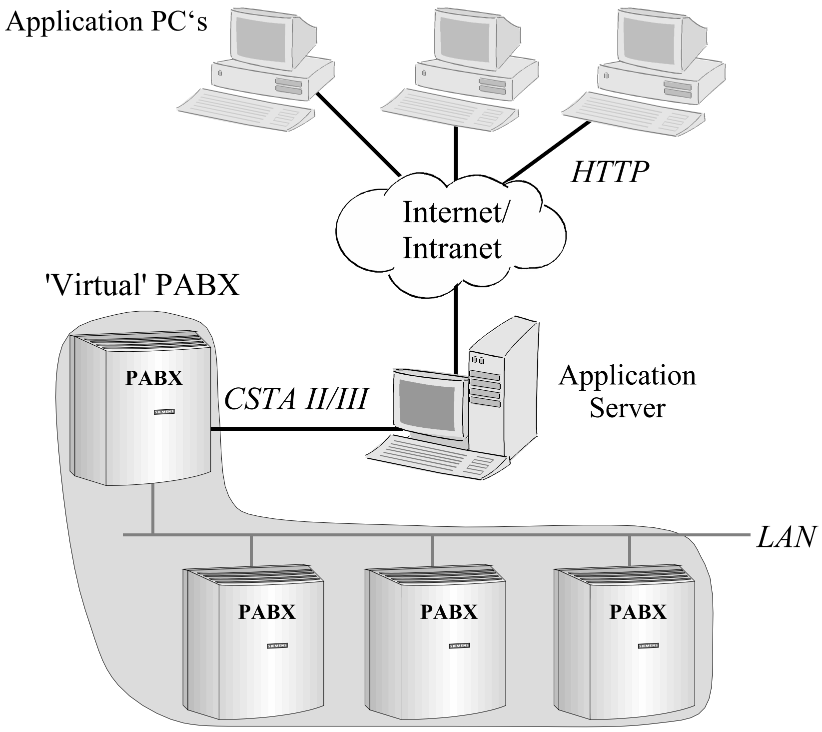


Figure 6.10: Setting for the Web-based Reconfiguration of Call Management via a Virtual PABX

The *Personal Call Manager* is embedded in an IP-based cluster of up to 4 midrange PABX of type *Hicom 150e*. The *Personal Call Manager* server is connected to a network of switches which are themselves connected through an IP-infrastructure. This multi-node solution can be seen externally as a *"virtual" PABX*: the system can grow and be reconfigured at runtime, and the handling of the up-to 250 users can be modularly organized over the physical switches. The general architecture of the considered scenario is illustrated in Figure 6.10. The "virtual" PABX is connected to an application server which also includes a web-server. The *Personal Call Manager* runs on the application server and is accessible via the web server. On the application server a *TAPI service provider* is also located which performs a mapping of the TAPI protocol on which the application and services are based, to the CSTA protocol.

To modify a call-forwarding for a specific port via the *Personal Call Manager* the user first selects the actual configuration via the application GUI. The application is then able to modify the configuration of the "virtual" PABX via the TAPI service provider, which sends the corresponding CSTA commands to the PABX. To configure a qualified call-forwarding that forwards calls to different numbers depending on several user defined time intervals, the *Personal Call Manager* actually modifies a call-forwarding for the specific port automatically every time the time interval changes.

6.4.2 Test Setting for the *Personal Call Manager*

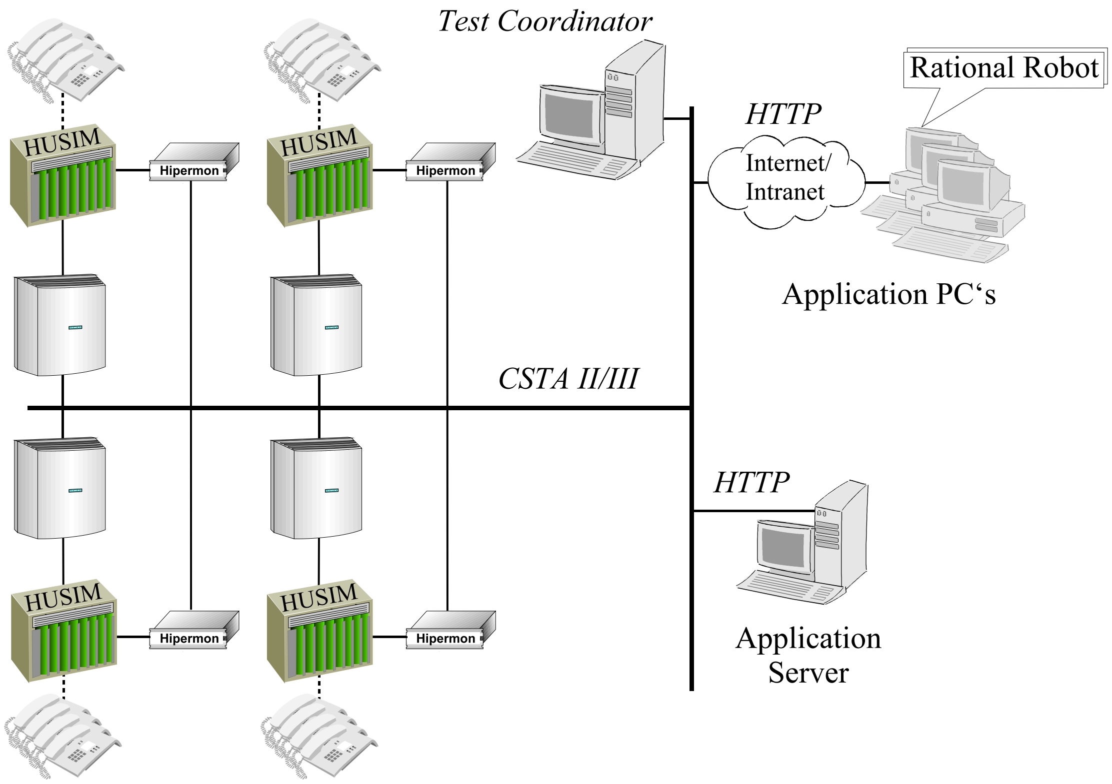


Figure 6.11: Architecture of the Test Setting for the *Personal Call Manager*

The concrete test setting for the *Personal Call Manager* is shown in Figure 6.11. We consider the full configuration here, i.e. a cluster of 4 PABX. Each PABX is tested by its an individual Hipermon/HUSIM combination, as used for the test of a single PABX, cf. Figure 6.6. For the test of the web-based application *Personal Call Manager*, several instances of the *Rational Robot* are used, each responsible for controlling a browser. Note that the application server is not considered in this test setting. The test of the *Personal Call Manager* application, however, implies some special testing properties:

— In the *Personal Call Manager* it is possible to specify a set of destination numbers which will be used sequentially as forwarding targets (in a sort of graceful exception handling) if the normally foreseen destination of the call-forwarding is either (i) busy, or (ii) not accessible, or (iii) has its own call-forwarding, or (iv) is itself controlled via the *Personal Call Manager*. Accordingly, it is important to test extreme situations where all forwarding destination numbers match the criteria mentioned above. In this case, the “default” case should be entered, i.e. no call-forwarding should be made.

— A second instance of this kind of problem occurs in conjunction with the specification of the forwarding time intervals. The user can specify different call-forwarding targets according to several time intervals, but it must always be ensured that a “default” case is defined for execution during the missing (i.e. implicitely undefined) time slots. Tests should ensure that (i) this default case will be entered during undefined time slots and that (ii) if no such default case is defined, no call-forwarding will occur.

— In the *Personal Call Manager* setting, a virtual PABX is composed of up to 4 physical PABX. Here each of the physical PABX is responsible for a subset of the “virtual” ports (i.e. it maps them to physical ports) but every switch works on the basis of the configuration for the whole “virtual” switch, i.e. knows the situation of all “virtual” ports. This implies that every change of the configuration of a (virtual) port needs to be distributed to all other connected switches to maintain coherence. Testing this mechanism requires testing (i) the exchanged protocol messages for the synchronization via the *Hipermon*, and (ii) the impact on the associated features.

— Frequent examples of admissibility criteria concern features that depend on the rights that a role allows. E.g. in the *Personal Call Manager* it is possible to create *profiles* for call-forwarding: they store the attributes of a qualified call-forwarding, i.e. time-intervals for the forwarding, the caller-id’s and the corresponding destination numbers. Obviously, access to this information underlies restrictions: only that particular user and an administrator can modify these profiles, but other selected users may have read-only access to the information. It is therefore important to test, the call-forwarding functionality itself, as well as that it is impossible for (normal) users to modify the profiles of other users. This results in fact new facets to the testing of telephone switches, since this kind of relation (access to resources in correspondence to roles/rights) only occurs during (re-)configuration processes.

6.4.3 Evaluation

For the test of the *Personal Call Manager* application 34 different test cases were designed and 46 new test blocks were implemented, again using the *Rational Robot*. Overall 586 instances of the test blocks were needed in the test cases (average number of test blocks per test case is approximately 17). The distribution of the used test blocks can be seen in Figure 6.12. In this test setting over 61% of test blocks could be reused. An average of approximately 6.5 internal test blocks are used in a test case. This is because of the relatively high number of involved test tools that needs to be initialized, e.g. when compared to the HPCO test setting. Furthermore when testing the *Personal Call Manager* application more PABX related test blocks are used.

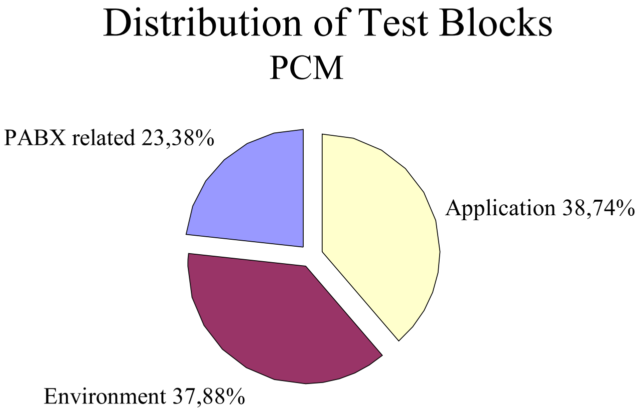


Figure 6.12: Distribution of used Test Blocks: PCM

6.5 Testing Miscellaneous CTI Solutions

We have also investigated the test of nine "smaller" CTI solutions, where "smaller" refers to the complexity of the used test setting. All solutions have in common that they take at most one application into account, which is sometimes even located on the application server. For this reason it is sufficient to test the application side with one instance of the *Rational Robot* only. In the remainder we briefly introduce three of the considered scenarios:

Hotel Solutions – Caracas Inn The *Caracas Inn* application is a PC based hotel solution for small and medium-sized hotels with a maximum of 250 rooms. It can be either installed on a single workstation or distributed over multiple ones. In cooperation with a *Hicom* switch, the main functions of *Caracas Inn* are: check-in resp. check-out, invoice management, name entry for caller identification, call charge recording and evaluation, room status management, recording of minibar use via telephone, etc.

Virtual Telephone – OptiPhone The *OptiPhone* is a virtual telephone that can replace a physical one. It is located on an application client and interacts with the PABX either directly via e.g. ISDN hardware in the client PC or the LAN, or by means of an application server via TAPI. The *OptiPhone* offers almost the full functionality of a "normal" telephone, e.g. basic calls, call forwarding, conference calls, redials, etc.

Call Charging Computer A *Call Charging Computer* is responsible for recording and assigning incoming and outgoing call charge data that permit evaluation by extension, trunk, department, etc. This can be done either passively, i.e. the PABX sends all call charge data to the *Call Charging Computer*, or actively, i.e. the *Call Charging Computer* gathers the information from the PABX. Nowadays a *Call Charging Computer* is normally realized as an application, running on a dedicated application server, which is connected to the PABX.

In addition we considered settings where e.g. voice mail or special CSTA clients were tested.

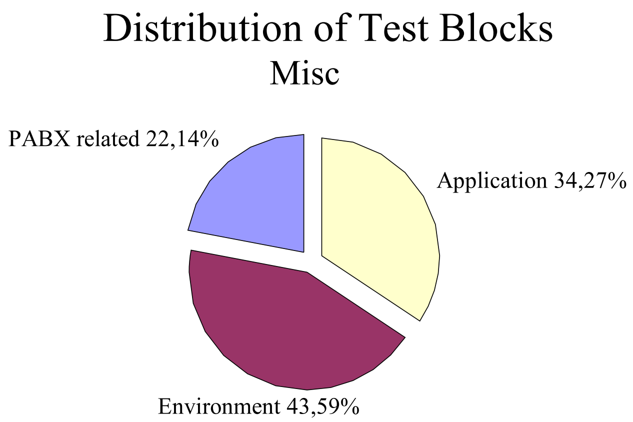


Figure 6.13: Distribution of used Test Blocks: Miscellaneous CTI Solutions

In total 109 test cases were developed for the test of all miscellaneous CTI solutions. Furthermore 97 application specific test blocks have been implemented and 1459 instances of test blocks are used in the test cases (an average of 13 test blocks per test case). The distribution of the test blocks is depicted in Figure 6.13. The reuse factor for these solutions is with almost 66% much higher than in the other settings, which emphasizes the assumption that “small” CTI solutions can be integrated into the *ITE* quite easily. Finally it must be noted that when converting the relatively high number of environmental test blocks into absolute numbers, we have discovered that less than 6 internal test blocks are needed per test case.

6.6 Evaluation

This chapter concludes with an evaluation of the *ITE* for the test of complex CTI solutions. We first discuss aspects concerning the design of the test cases, and afterwards the economic impact of the *ITE* with respect to selected applications.

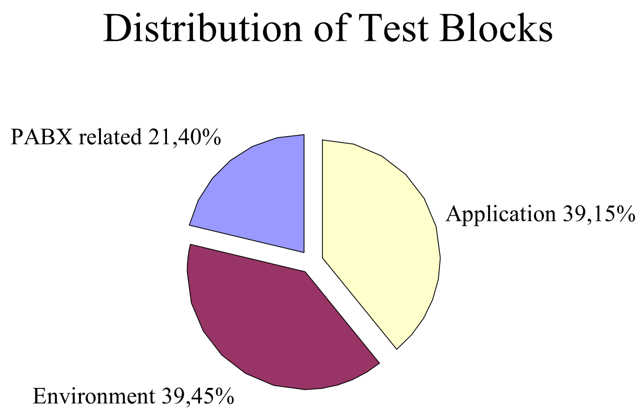


Figure 6.14: Distribution of used Test Blocks: Overall

The total number of test cases for all investigated CTI solutions is 314, where 302 different test blocks were implemented. These test blocks occur 4416 times, which makes an average of 14 test blocks per test case. The reuse factor lies above 60%, where each PABX related test block is used an average of 30 times, each environmental one 53 times and each application specific 7 times. So even the test blocks that are implemented for a particular test setting can be reused. The overall distribution of used test blocks is illustrated in Figure 6.14. It must be noted that an average of twice as many test blocks concerning the applications are used as PABX related. The reason for this is that the PABX has already tested intensively individually, so that the tests within the *ITE* can really concentrate on the interaction between the PABX and the CTI applications.

Table 6.1: Regression Test Cost factors

Task	manual	with *ITE*	frequency
Test planning	$\surd$	$\surd$	once
Test specification	$\surd$	$\surd$	once
Test scripts	($\surd$)	$\surd$	once
Test execution	$\surd$	-	recurrent
Test protocol	$\surd$	-	recurrent
Test analysis	$\surd$	($\surd$)	recurrent

To evaluate the economic impact of the *ITE* introduction, we must first identify the cost factors that pertain to testing CTI systems. Table 6.1 identifies the macroscopic cost factors that arise during the lifecycle. They are listed together with their frequency of occurrence and with a qualitative indication of their relevance in a manual testing and an automated testing scenario. Test planning and specification and the definition and setup of test scripts occur only initially, when an experimental scenario (i.e. the testing of a specific CTI system) is set up. The planning and specification phases are not affected by the *ITE*. The usual collection or programming of test scripts that directly constitute the elementary test blocks in a manual setting is in the *ITE* additionally supported by a largely automated generation of reusable test blocks that fit in with the overall *ITE* architecture. The additional effort required by this wrapping is compensated for the increased reuse and ease of test design, but it does require some additional effort.

The main focus of the *ITE* is, however, the reduction of costs for the repetitive, recurrent phases of CTI testing (cf. *Test Phase* of the test process defined in Section 3.3.2): primarily, we address the test execution (cf. Table 6.2), in combination with the automatic creation of test reports.

Table 6.2 documents the measured improvement of the test execution costs due to the introduction of *ITE* at Siemens. The systems under test considered in each row are composed by the PC client-server application listed in Col. 1 cooperating with the *Hicom* switch along the configuration pattern illustrated in Figure 6.6[3]. The second and third column report the measured effort (in man hours) of one regression cycle for the system under test when performed manually (Col. 2) or with the *ITE* (Col. 3). The improvement is dramatic: factors between 10 and 25 for each regression cycle execution. The full automation is not yet feasible at the moment since some manual steps like system setup and configuration (e.g. physical connection of the components, installation of the software on the machines) are still needed.

[3]Note that because of still incomplete data we were not able to evaluate all of the previously discussed test settings.

Table 6.2: Test execution effort in hours per regression

System-under-test	manual	with ITE
Hotel Solutions	10,0	0,5
Call Center Solutions	43,0	1,0
Analog Voice Mail	23,0	0,5
Digital Voice Mail	20,0	0,5
Call Charge Computer	19,0	0,5
Total	115,0	5,0

These initial results are indeed representative for average behaviour. Concerning the next applications that are joining the *ITE*, the conservative expectations shown in the last rows of Table 6.2 also indicate a factor of approximately 20.

A global cost-benefit calculation shows that the additional investment for *ITE* is well able to pay off in a short period of time, if extensively adopted. In fact the *ITE* dramatically reduces the recurring cost factors, without increasing the remaining positions significantly which are concerned with the basic effort that still has to be made along the whole test lifecycle (test planning, manual configuration of the test settings, ...) and the necessary upfront investments (e.g. licence fees for test tools, hardware, ...).

Chapter 7

Testing Web-based Applications

Modern *web-based applications* are complex multitiered, distributed applications that usually run on heterogeneous platforms. For testing purposes it is not sufficient to concentrate on the analysis of the static structure of a web site alone, as today's web-based applications must be treated as applications which are particularly characterized by a high degree of (multiple) user interactions and the presentation of dynamic content. This has a huge impact on the test and validation requirements: web-based applications have to be handled as complex systems. So testing these kind of systems fits well into the *ITE* approach.

We will first introduce the considered application domain, i.e. web-based applications (Section 7.1). After that we discuss in Section 7.2 an appropriate test architecture for testing such applications, together with a concrete realization of the *ITE*. Finally in Section 7.3 we present two case studies made by testing web-based applications: the test of the *Online Conference Service* [NMS02, MNS02a, MNS02b] and the *Bug Tracking System* [Raf02].

7.1 Web-based Applications

A *Web-based Application* is an URL addressable resource returning information in response to client requests. A *Uniform Resource Locator* (*URL*) [Wor94], or more generally a *Uniform Resource Identifier* (*URI*) [Wor99c], provides a standard way of expressing the location and data type of a resource. URL's in general take the form *protocol://address*, where *protocol* is e.g. *HTTP* or *FTP*, and the address is merely the server and pathname of a given resource. Figure 7.1 sketches the typical architecture of a web-based application. The browser, located on a client, plays the role of the "classical" application GUI and interacts with a *Web server* via the *Hypertext Transfer Protocol* (*HTTP*) [Wor99b], which is an application-level protocol for

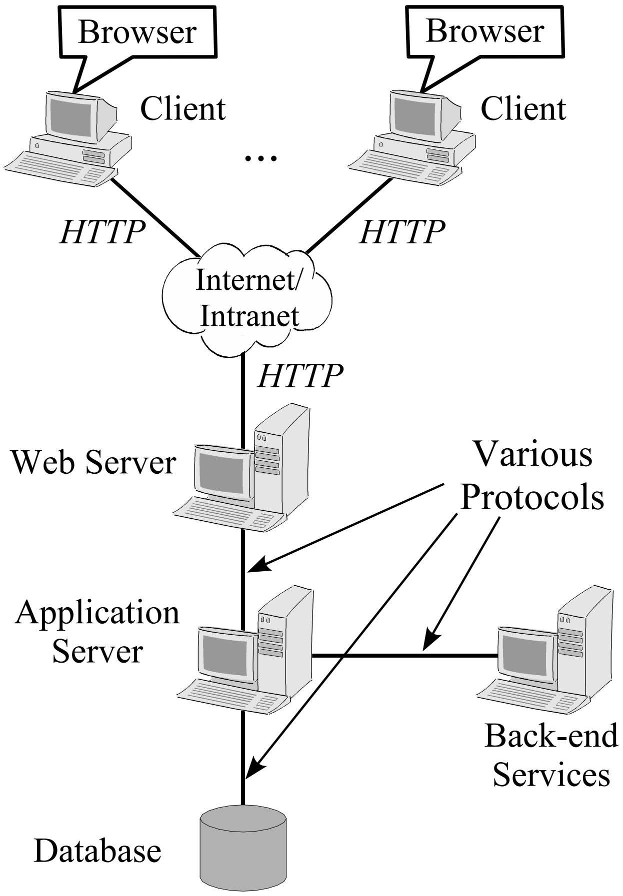


Figure 7.1: Overview of the Architecture for a Web-based Application

distributed, collaborative, hypermedia information systems. The web server itself communicates with an *Application Server* that dynamically builds the requested web pages by means of an interaction with a *Database* and (several) *Back-end Services*. Here various protocols are often used, e.g. among others CORBA, RMI, or database access protocols like *JDBC* or *ODBC*. The architecture of Figure 7.1 is somewhat simplified, as staging servers, firewall solutions, load balancing and redundant architectures often increase the speed and availability of the offered application, but also increase the complexity of the considered architecture.

The Hypertext Transfer Protocol

HTTP is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers. A feature of

HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.
The HTTP protocol is a request/response protocol. A client sends a request to the server in the form of a request method, URI, and protocol version, followed by a message containing request modifiers, client information, and possible body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code, followed by a message containing server information, entity meta-information, and possible entity-body content.

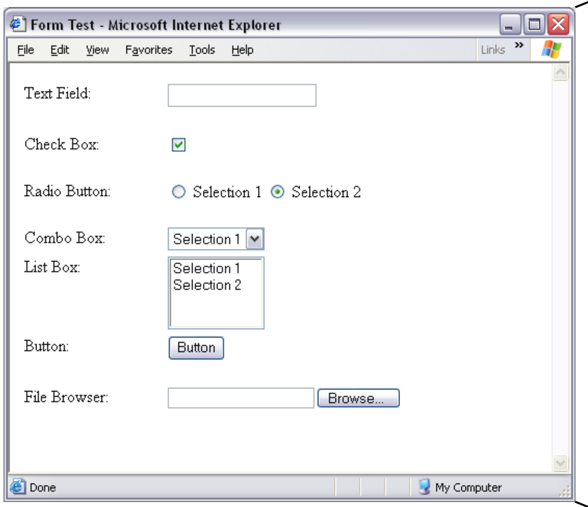
The Hypertext Markup Language

The pieces of information that are provided by a web-based application are exchanged in the *Hypertext Markup Language* (*HTML*) [Wor99a]. HTML is an SGML application, i.e. defined by a DTD. Note that for each version of HTML a different DTD exists.

```
<!DOCTYPE HTML PUBLIC ...>
<HTML>
  <HEAD>
    <TITLE>...</TITLE>
    ...
  </HEAD>
  <BODY>
    ...
  </BODY>
</HTML>
```

Figure 7.2: Structure of a HTML document

First of all a HTML document is made up of a section where the document type (and a version) is specified, i.e. the corresponding DTD is stated, which defines both type and version. Furthermore HTML documents are structured into two parts: the *HEAD* and the *BODY*. The *HEAD* contains general information, or meta-information, about the document. The *BODY* element contains all the content of a document. Various mark-up elements are allowed within the body to indicate headings, paragraphs, lists, tables, hypertext links, images, and so on. Additional it is possible to specify forms in HTML, so that an interaction with a user can take place. A HTML form is the section of a document containing normal content, markup, special elements called controls (checkboxes, radio buttons, menus, etc.), and labels on those controls. Users generally "complete" a form by modifying its controls (entering text, selecting menu items, etc.), before submitting the form to an agent for processing (e.g., to a web server, to a mail server, etc.). In Figure 7.3 all available controls of a form are presented in a browser (Figure 7.3 (left)) and for each a portion of the corresponding HTML source (Figure 7.3 (right)).




```
...
<INPUT type="text">
...
<INPUT type="checkbox">
...
<INPUT type="radio" name="name"
       value="Selection 1">
<INPUT type="radio" name="name"
       value="Selection 2">
...
<SELECT size=2>
    <OPTION>Selection 1</OPTION>
    <OPTION>Selection 1</OPTION>
</SELECT>
...
<SELECT size=2 multiple >
    <OPTION>Selection 1</OPTION>
    <OPTION>Selection 1</OPTION>
</SELECT>
...
<INPUT type="button">
...
<INPUT type="file">
...
```

Figure 7.3: Controls of Forms in HTML

Text Field Two types of controls are available to allow users to input text: a simple text field (single-line input control) and a text area (multi-line input control).

Check Box A check box is an on/off switch that may be toggled by the user.

Radio Button Radio buttons are like check boxes except that when several share the same control name (cf. Figure 7.3 (right)), they are mutually exclusive: when one is switched on, all others are switched off.

Combo Box A combo box or menu offers users options to choose from.

List Box A list box is like a combo box except that several options can be chosen simultaneously.

Button There exist three types of buttons:

— *submit* buttons are used to submit the form;

— *reset* buttons reset all controls to their initial value and

— *push* buttons have no default behaviour and can trigger client-side script, e.g. programmed in the *Java Script Language*.

File Browser This control allows the user to select (local) files, so that their contents may be submitted with the form.

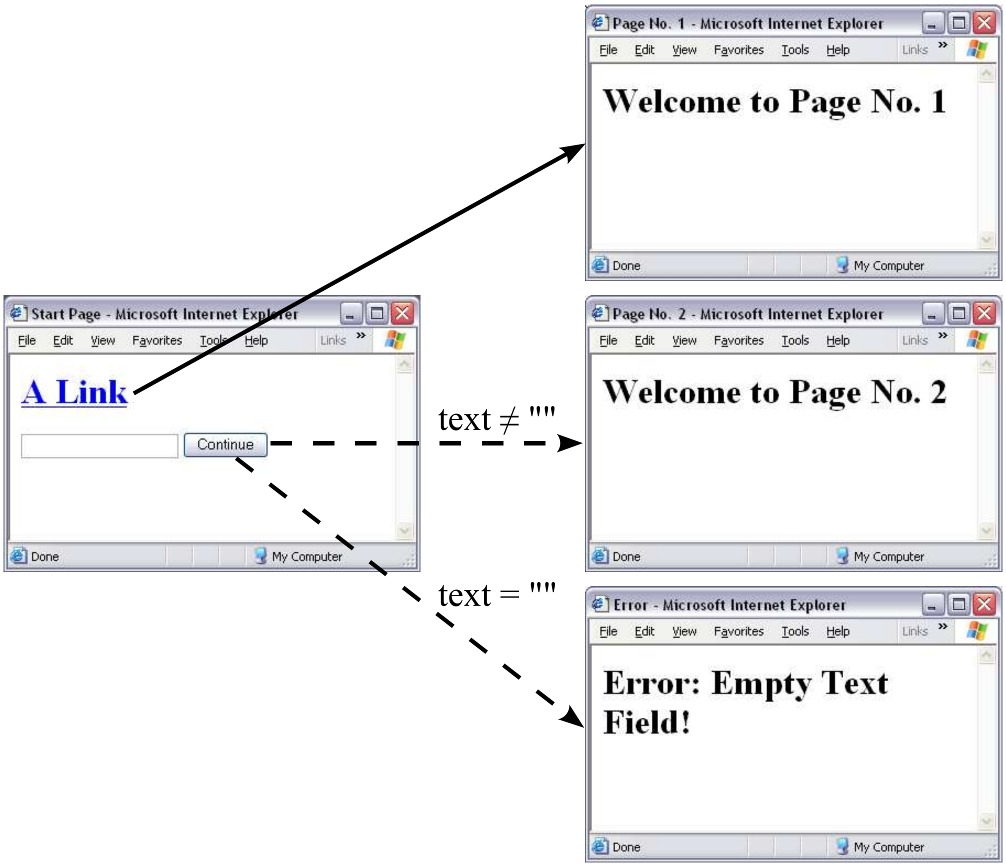


Figure 7.4: Dynamic Behaviour of HTML Sites

In Figure 7.4 the dynamic behaviour of an HTML site, i.e. a collection of coherent HTML documents, is indicated. Note that an HTML site is usually stored as a set of static HTML pages on a web server. Starting at a dedicated HTML document the user can control the subsequent execution flow, i.e. HTML documents shown in the browser. Here we can distinguish between two different types of control: *static links* (solid line in Figure 7.4) and *forms* (dashed line). Whereas for the first the target is fixed within a HTML document, for the latter the succeeding HTML document depends on the content of the form filled in by the user. For instance, in Figure 7.4 depending on the value of the text field, different HTML documents will be presented to the user. Consequently the static structure of an HTML site can be computed in advance, i.e. in the form of a dependency or link graph by computing the reflexive, transitive closure of links, starting at the start page. The static structure of an HTML site is often referred to as a *Site Map*. Forms, however, are usually evaluated on the side of the server and the subsequent execution flow can therefore not be computed in advance. Note that form evaluation is usually not as simple as depicted in the situation of Figure 7.4.

7.2 System-level Testing of Web-based Applications

Contemporary web application testing tools are still dominated by static approaches, focussing on a posteriori structure reconstruction and interpretation [RT01, LKHH00] rather than on functional aspects directly related to the applications design. As mentioned above web-based applications generate HTML documents dynamically on demand. This implies that a site map cannot be computed in advance, because the HTML documents are not directly stored on a web server. Therefore web-based applications must be treated as applications and it is not sufficient to restrict the testing to pure link testing or performance issues. Due to the architecture of web-based applications, the *ITE* is predestined for testing such systems.

In Figure 7.5 the test architecture for web-based applications within the *ITE* is depicted. Here the test coordinator has access to the client PC's, or more precisely to the browsers located on the client PC's, via the *Rational Robot*. This enables us to test even complex workflows of web-based applications, where the interplay between various involved users must be tested. Usually different users have different rights resp. permissions. Here it is generally not sufficient to use several instances of a browser on a single client only, as they usually share common session information, so that it is not possible e.g. to login with different identities. With the *ITE*, however, it is possible to distribute the treatment of different users over different clients, as it occurs in reality. Furthermore, within the *ITE* approach it is possible to take the server side into account as well, i.e. denoted by the dashed test tools, cf. [NMS02, MNS02a]. However, we will concentrate here on the test of multiple clients only, where the following properties are of particular interest:

Interdependencies Interdependencies between the various involved users have to be tested intensively, because they usually work on shared data.

Safety Criteria Here the rights and permissions of a user must be tested, e.g. to ensure that sensitive data is not accessible without having the corresponding permissions assigned.

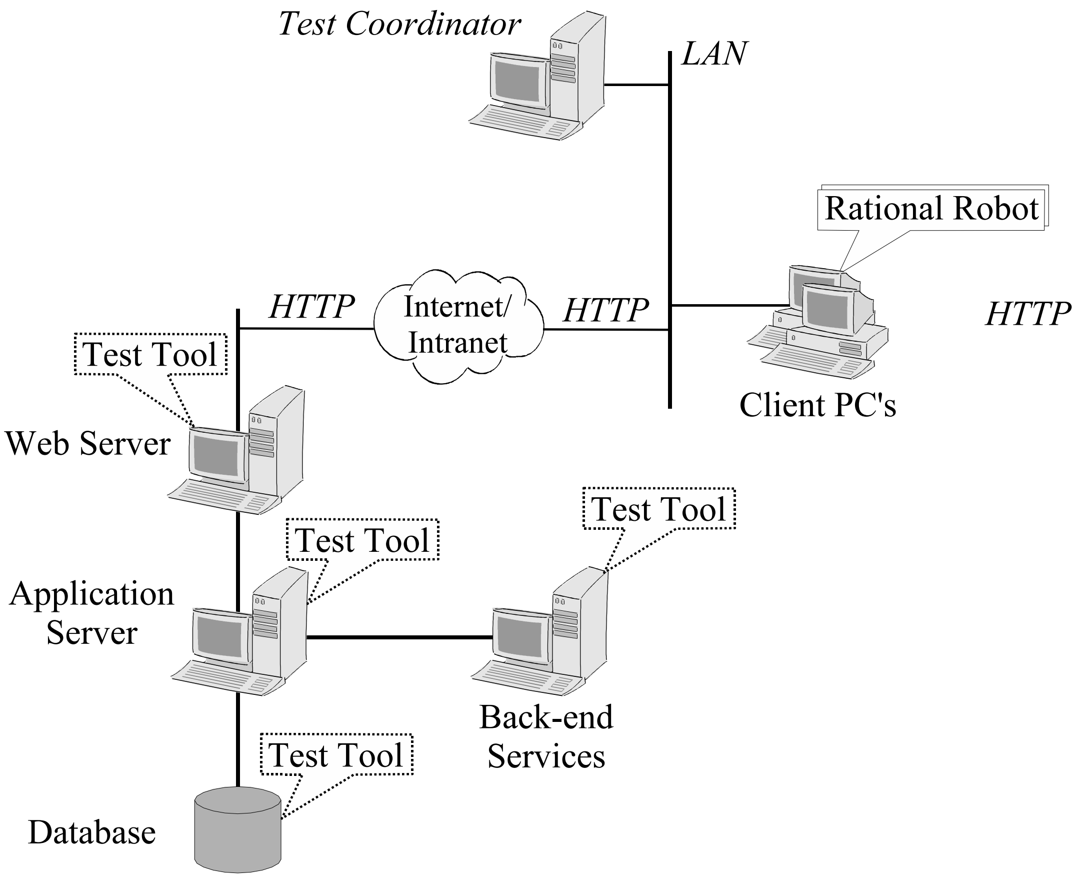


Figure 7.5: Test Architecture for a Web-based Application

Admissibility Criteria Because of the highly distributed character of web-based applications, i.e. due to the involvement of a whole web-server architecture, a test for the admissibility of single operations is needed.

For the test of the browser located on the client PC's we will again use the *Rational Robot*. The *Rational Robot* provides sophisticated support for testing web-based applications. It is possible to record a test script with one browser (e.g. *Internet Explorer*) and to replay it with another browser without changes (e.g. *Netscape Browser*). Furthermore, the *Rational Robot* offers verification points especially suited for testing the elements of HTML documents, e.g. checking of hyperlinks, images, tags, content, tables, etc.

The restricted and specially standardized set of available GUI controls simplifies the test of web-based applications. This is why in contrast to the test of "normal" applications, where customized GUI controls can be used, once a set of test blocks has been implemented it can be used for the test of almost every web-based application (from the client point of view). Table 7.1 shows the test blocks that have been implemented in the *ITE* for supporting the test of web-based applications. They can be divided into three different sections: internal action test blocks for the handling of a browser, external action check test blocks to command the HTML controls, and external check test blocks for checking certain properties of HTML documents[1]. Please note that as a file browser control is composed of a text field and a button, it can be handled via the text field resp. button-related test blocks.

Table 7.1: Test Blocks for Testing Web-based Applications

`startBrowser`	Starts a browser on a client.
`stopBrowser`	Stops a browser on a client.
`initBrowser`	Initializes a browser. Here in particular the currently shown page will be cleared.
`setTextfield`	Inputs text in a text field.
`setCheckBox`	Activates resp. deactivates a check box.
`setRadioButton`	Activates a radio button.
`setComboBox`	Selects the entry of a combo box.
`setListBox`	Selects an (additional) entry of a list box.
`pressButton`	Presses a button.
`followLink`	Follows a link.
`gotoURL`	Redirects the browser to an absolute URL.
`checkURL`	Checks the actual URL of the currently shown page.
`checkTitle`	Checks the actual title.
`checkContent`	Checks whether the given text is visible on the actual page.
`checkCheckBox`	Checks the status of a check box.
`checkRadioButton`	Checks the status of a radio button.
`checkComboBox`	Checks whether a combo box contains an entry.
`checkListBox`	Checks whether a list box contains an entry.

7.3 Testing various Web-based Applications

We have done two different case studies with the test of web-based applications: the test of the *Online Conference Service* and the test of the *Bug Tracking System*.

The Online Conference Service [LMS01] is a complex web-service that supports online the management of the scientific program of professional conferences. It

[1] Although HTML documents can be checked with the available check test blocks, it is often useful to extend the check test blocks for checking application specific properties.

proactively helps *authors*, *Program Committee chairs*, *Program Committee members*, and *reviewers* to cooperate efficiently during their collaborative handling of the composition of a conference program. The application provides a timely, transparent, and secure handling of the papers and of the related submission, review, report and decision management process. The online conference service is a powerful application, including a role-based access-and-rights-management feature that is reconfigurable online, which leads to high flexibility and administration comfort, but which is responsible for potentially disruptive behaviour in connection with sensitive information. Because of the complexity of the underlying workflows (and of course, therefore, the complexity of the application itself) intensive testing is clearly necessary. The test for the submission of an article to a conference demonstrates how complex the process is to ensure that an article submitted by an author, is correctly delegated to a member of the program committee. From the submission to the start of the review process of the article at least three different roles are involved: the author himself; the conference chair, who must delegate the article to a member of the program committee; and finally the members of the program committee, who are responsible for the reviews. So we need at least three concrete users to run this test, whereby each user has a distinct role. Note that the three users are distributed over three clients.

The second web-based application considered here is the *Bug Tracking System* of [MET], which organizes the administration of bugs in software projects. In contrast to the conference service, the functionality of the bug-tracking system is not implemented in the web-based application itself, but delegated to the *Gnats* system [Gna], a set of tools for tracking bugs reported by users to a central site. Gnats allows problem report management and communication with users by various means. Gnats stores all the information about problem reports in its databases and provides tools for querying, editing, and maintenance of the databases. The bug tracking system provides a web interface for accessing the information offered by the Gnats system. Interesting test purposes are e.g. to test whether a bug report, created by an arbitrary user and eventually marked as confidential, can be accessed (read only) by another user. A similar test purpose is to check whether a test report can be deleted, which should only be possible for administrator users, or edited by developers to document the current status of the bug.

Summarizing, we have investigated two different kinds of web-based applications, i.e.

1. The online conference service, which is a pure web service, i.e. all functionality is implemented in the application itself. No additional back-end server is needed for the realization of this service.

2. The bug tracking system, which us based on an existing legacy system (Gnats), located on an back-end server. Here the web-based application provides a sophisticated frontend for using Gnats, but adds no additional functionality.

It is obvious, that the *ITE* instance for testing web-based applications is generic enough so that for both concrete test settings no further integration effort was required, i.e. in terms of the integration of new test tools or the realization of additional test blocks. We have, however, restricted the testing activities to testing the client side only, which perfectly fits into the black box testing approach. There are testing purposes that need a more sophisticated test setting, i.e. that take the back end side into account as well. Examples of this kind of applications can be, e.g.

E-Commerce When testing E-Commerce applications (e.g. Web-shops) it is important to consider simple functional requirements like *a user can check the shopping cart after buying an item* or security/administrative requirements like *a user can be logged in only once*. But more complex tests must also be performed and these take the whole distributed configuration of the service into account e.g. *check the validity of a credit card*. Here it is no longer sufficient to perform tests that consider the client side only, since e.g. a faulty implementation of a back-end service may accept all credit cards without checking them correctly.

(Safety critical) Web-services A good example of this special kind of Web-services is e.g. *Online/Internet-Banking*. It is of crucial importance that on the one hand the user-interface reacts as prescribed and that on the other hand the back-end services are interacting correctly with the clients, as in this scenario there are normally well established back-end services or legacy systems in use. In this special field of applications every fault (either functional errors or a system crash) can be extremely expensive in terms of indemnification, and, more importantly, may cause a loss of trust.

As indicated in Figure 7.5 by the dashed test tools, the *ITE* approach supports these enhanced test settings as well. It must, however, be noted that in these cases an additional, application specific integration resp. setup effort is required.

Part IV

Improvement Opportunities

Chapter 8

A Posteriori Generation of Approximate Models

In this chapter we present an approach for generating approximate models for complex systems a posteriori. Such models can never be exact, i.e. reflect the complete and correct behaviour of the considered system. Nevertheless they can be useful in practice, to present the cumulative knowledge of the system in a consistent description. Such knowledge consists of e.g. observations of the system, (partial) specifications, and expert knowledge. In Chapter 6 we have discussed that particularly in the telecommunication area, revision cycle times are extremely short, making the maintenance of specifications unrealistic, and at the same time the short revision cycles necessitate extensive testing effort. All this could be dramatically improved if it were possible to generate and then maintain appropriate reference models steering the testing effort and helping to evaluate the test results.

By optimizing a standard learning method according to domain-specific structural properties, we are able to generate approximate models for complex reactive systems. Learning is only feasible if one can check actively whether a given abstract sequence corresponds to (is an abstraction of) a concrete system behaviour. In fact it was the *ITE* that enabled us to implement learning procedures in practice by bridging the gap between the abstract models and the "real" world. Its flexible test specification formalism supports the generation of concrete test cases from abstract propositional sequences. Furthermore, its precise test execution semantics ensures that we can perform the mapping of the results of the test runs back into the abstract sequences so that they can be processed by the learning algorithm. We emphasize the practicability of our method by means of experiments we have carried out in an industrial setting.

The outline of this chapter is as follows. After a motivation (Section 8.1), in Section 8.2 we present the underlying basic learning algorithm. Section 8.3 discusses the application-specific adaptations, needed to implement a learning algorithm in practice with the help of the *ITE*. Furthermore, we present several improvements of the basic algorithm:

— A first improved version of the algorithm L^* is introduced in Section 8.4. It elaborates on the application-specific characteristics of the considered scenario, i.e. prefix-closeness, input-determinism, as well as partial order and symmetries between events. We have implemented filters that helps to improve the algorithm by incorporating the above mentioned characteristics. This way we do not need to change the core algorithm itself and are able to adapt the learning procedure easily to new settings, by selecting the appropriate filters according to the system characteristics.

— In Section 8.5 we provide an adaptation of the core learning algorithm itself called $L^*_{i/o}$. It is is tailored for dealing with input/output deterministic systems. Therefore we have changed the general datastructure of the algorithm, i.e. the observation table. Due to this changes, some of the system characteristics are already incorporated into the algorithm itself (prefix-closeness and input-determinism), while others are still implemented via filters (partial order and symmetries between events). Additionally we learn the systems in a representation that is tailored for input/output deterministic systems and are able to improve the performance once more.

8.1 Motivation

The aim of our work is improving quality control for reactive systems as can be found e.g. in complex telecommunication solutions. A key factor for effective quality control is the availability of a specification of the intended behaviour of a system or system component. In current practice, however, precise and reliable documentation of a system's behaviour is only rarely produced during its development. Revisions and last minute changes invalidate design sketches, and while systems are updated in the maintenance cycle, often their implementation documentation is not. It is our experience that in the telecommunication area, revision cycle times are extremely short, making the maintenance of specifications unrealistic, and at the same time the short revision cycles necessitate extensive testing effort (cf. Chapter 6). All this could be dramatically improved if it were possible to generate and then maintain appropriate reference models steering the testing effort and helping to evaluate the test results. In [HHNS02] it has been proposed to generate the models from previous system versions by using learning techniques and incorporating further knowledge in various ways. We call this general approach *moderated regular extrapolation*. Motivation for this name comes from the fact that learning in this domain involves generalizing from finite observations to cyclic behaviour patterns, which comprise infinite behaviour. The approach is tailored for a posteriori model construction and model updating during the system's lifecycle. The general method includes many different theories and techniques.

Regression testing provides a particularly fruitful application scenario for using extrapolated models. Here, previous versions of a system are taken as the reference for the validation of future releases. By and large, new versions should provide everything the previous version did. I.e., if we compare the new with the old, there should not be too many essential differences. Thus, a model of the previous system version could serve as a useful reference to much of the expected system behaviour, in particular if the model is abstract enough to focus on the essential functional aspects and not on details like, for instance, exact but irrelevant timing issues. Such a reference could be used for:

— Enhanced test result evaluation: Usually, success of a test is measured only via very few criteria. A good system model provides a much more thorough evaluation criterion for success of a test run.

— Improved error diagnosis: Related to the usage above, in case of a test error, a model might expose already very early some discrepancy to expected behaviour, so that using a model will improve the ability to pinpoint an error.

— Test-suite evaluation and test generation: As soon as a model is generated, all the standard methods for test generation and test evaluation become applicable (see [LY96, BT00] for surveys about test generation methods).

For a detailed discussion of further usages of the extrapolated models please refer to [HHNS02].

In this chapter we will focus on the method's fundamental ingredient, learning a finite system model from observations of the real system, from a practical point of view: How it is possible to successfully exploit automata learning, a currently mostly theoretical research area, in an industrial setting for the testing of telecommunication systems.

Learning and testing are two wide areas of very active research – but with a rather small intersection so far, in particular with respect to practical application. At the theoretical level, a notable exception is the work of [PVY99, GPY02]. There, the problem of learning and refining system models is studied. The ultimate goal may be testing a given system for a specific property, or correcting a preliminary or invalidated model. In contrast our approach focuses on the very practical aspects of learning models of real-life systems based on the concepts of *moderated regular extrapolation* [HHNS02].

8.1.1 Moderated Regular Extrapolation

In [HHNS02] we propose a new method for model generation, called (moderated) *regular extrapolation*, which is tailored for a posteriori model construction and model updating during the system's lifecycle. The method, which comprises many different theories and techniques, makes formal methods applicable even in situations where no formal specification is available: based on knowledge accumulated from many sources, i.e. observations, test protocols, available specifications and last but not least knowledge of experts, an operational model in terms of an (input/output) deterministic finite automaton is constructed that uniformly and concisely resembles the input knowledge in a way that allows further investigation.

A key feature of our approach is the largely automatic nature of the extrapolation process. The main source of information is observation of the system, represented by system traces. These traces may be obtained passively by profiling a running system, or they may be gathered as reactions of the system to external stimulation (as in testing). Extrapolation from passively obtained observations and protocols of test runs may yield a too rough model of the system, leaving out many of its features and generalizing too freely. So these models have to be refined. We adapt machine learning algorithms and also incorporate expert's knowledge. Learning consists of running specific tests with the aim of distinguishing superficially similar states and finding additional system traces. Experts can either be implementors or people concerned with the environment of the system, for instance people knowing the protocols to be observed. Their knowledge enters the model in the form of declarative specifications, either to rule out certain patterns or to guide state distinguishing searches. Technically this is done by employing the bridge from temporal logic to automata, model checking techniques and partial order methods. Conflicts arising during the extrapolation process between the different sources of information have to be examined and resolved manually (moderation).

The starting point of our investigation was the regression testing problem in the so-called black box scenario: a technique was needed to deal with a large legacy telephony system in which most of the involved applications (the so-called "value-added" products) running on and with the platform are third party. There was no hope of obtaining formal specifications (cf. Chapter 6). The only source of information was intuitive user manuals, interaction with experienced test engineers and observations, observations, observations. As none of these sources could be fully trusted (and since what is true today may not be true tomorrow), the only possible approach was to faithfully and uniformly model all the information and to manually investigate arising inconsistencies. This lead to a change management process with continuous moderated updates.

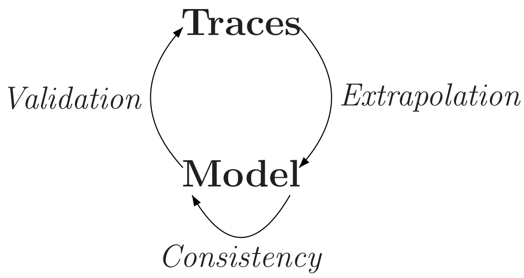


Figure 8.1: Generation of Models

Fig. 8.1 sketches briefly our iterative approach. It starts with a model (initially empty) and a set of observations. The observations are gathered from a reference system in the form of traces. They can be obtained either *passively*, i.e. a running reference system is observed, or *actively*, i.e. a reference system is stimulated through test cases. The set of traces (i.e. the observations) is then preprocessed, extrapolated and used to extend the current model. After extension the model is completed through several techniques, including adjusting to expert specifications. The last step validates the current hypothesis for the model, which can lead to new observations.

It is impossible in practice to find a precise model of the system under consideration, even on an abstract level. Such a model would usually be far too large and, as results from learning theory indicate, too time-consuming to obtain and to manage. Instead we are aiming at concise, problem-specific models, expressive enough to provide powerful guidance and to enhance the system understanding. It is of course very important that the models are reasonably close to the system. Exploiting all the information at hand, independently of their source, to obtain a comprising "hypothesis" model is the best we can do.

8.2 L^*: A Basic Learning Algorithm

In the domain of machine learning, beyond others, the problem of constructing an acceptor, in terms of an automata representation, for an unknown regular language is studied. If the result is to be exact, and the only means of information are tests for membership in the considered language and a bound on the number of states of the minimal accepting deterministic finite automaton (DFA) for the language, the worst-case complexity is exponential in the number of states of the acceptor [Moo56]. If other sources of information are available, i.e. an equivalence test whether a hypothesis is a correct acceptor for the unknown language, the learning procedure can be done in polynomial time[1] with the algorithm L^* [Ang87]. Furthermore, the equivalence test enables the algorithm to achieve an exact result, not only an approximation. Thus for the concrete realization of the model generation procedure, we propose to use the machine learning algorithm L^* of [Ang87] as a basis.

8.2.1 Introduction to L^*

Angluin describes in [Ang87] a learning algorithm for determining an initially unknown regular set exactly and efficiently. To achieve this aim besides the alphabet A of the language two additional sources of information are needed: a *Membership Oracle* (MO) and an *Equivalence Oracle* (EO). A *Membership Oracle* answers the question whether a sequence σ is in the unknown regular set or not with `true` or `false`. The *Equivalence Oracle* is able to answer the question whether a *hypothesis* (an acceptor for a regular language) is equivalent to the unknown set with $\perp$ or a counterexample, where $\perp$ denotes that there exists no counterexample, which implies that the *hypothesis* accepts the set to be learned. The hypothesis is represented as a deterministic finite automaton.

Definition 8.1. A *deterministic finite automaton* ($\mathcal{DFA}$) is a structure $\mathcal{S} = (\Sigma, A, \delta, s_0, F)$, where

— Σ is a finite, non-empty set of states,

— A is a finite set of actions,

— δ denotes the transition function $\delta : \Sigma \times A \to \Sigma$,

— s_0 is the unique start state,

— F is the non-empty set of accepting states $F \subseteq \Sigma$.

We write $s \xrightarrow{a} s'$ if δ maps (s, a) to s', and extend this notation to strings $\sigma \in A^*$. The language $\mathcal{L}(\mathcal{S})$ accepted by the automata is the set of strings which lead to an accepting state, i.e. $\{\sigma \in A^* \mid \exists s \in F.\, s_0 \xrightarrow{\sigma} s\}$.

The basic idea behind Angluin's algorithm (cf. Algorithm 8.1) is to systematically explore the system's behaviour using the membership oracle and trying to build the transition table of a deterministic finite automaton with a minimal number of states. The algorithm starts with the initial state, which is reached by the empty

[1] Polynomial in the number of states of the minimum acceptor and the maximum length of any counterexample [Ang87].

string. It keeps a set of strings S which lead from the initial state to all the states discovered so far. S may, and in fact usually will, contain for several states more than one string leading to it. It also maintains a set E of strings which distinguishes between states accessed by S in that some e is accepted after one element of S but not the other. With membership queries, the algorithm tests whether all states reachable in one step from the states so far behave like known states (*closed* table in the following definition). This provides a guess for the transition table of the automaton. It is also checked whether all strings which lead to states thought to be equivalent so far have the same one-step behaviour (*consistent*). If not, S or E are extended according to the observed discrepancy. Otherwise (if everything seems correct), an equivalence query is raised with an automaton constructed from the available information, i.e. the actual observation table. If the query is not successful, a counterexample is returned in the form of a string which serves to distinguish further states and another iteration will be started.

The central data structure of the algorithm is called *observation table* $\mathcal{OT}$, where the results of membership queries are stored and from which a hypothesis is read off.

Definition 8.2 (Observation Table). An observation table for an alphabet A is a triple $\mathcal{OT} = (S, E, T)$, where $S \subset A^*$ is a finite, prefix-closed set, $E \subset A^*$ is a finite, suffix-closed set, and T is a finite function mapping strings of $(S \cup S \cdot A) \cdot E$ to entries out of $\{0, 1\}$. Furthermore if s is an element of $(S \cup S \cdot A)$, then $row(s)$ denotes the finite function f from E to $\{0, 1\}$ defined by $f(e) = T(s \cdot e)$.

1. $\mathcal{OT}$ is called **closed**, if $\forall t \in S \cdot A.\ \exists s \in S.\ row(t) = row(s)$.
2. $\mathcal{OT}$ is called **consistent**, if $\forall s_1, s_2 \in S.\ row(s_1) = row(s_2) \Rightarrow$
   $\forall a \in A.\ row(s_1 \cdot a) = row(s_2 \cdot a)$.

Fact 8.3. Obviously if two rows $row(s)$ and $row(s')$ are not equal, there exists at least one element e out of E such that $T(s \cdot e) \neq T(s' \cdot e)$.

8.3 Application-Specific Adaptations to L^*

The considered application domain is the functional regression testing of complex, reactive systems, as discussed in Section 3.2. In this section we first present the application-specific adaptations of L^* so that it can be used for learning models for reactive systems. We start with a discussion of some assumptions about the considered application domain and the resulting implications for the learning procedure

Algorithm 8.1: L^*

Alphabet A, Observation Table $\mathcal{OT} = (S, E, T)$, where initially $S = E = \{\lambda\}$

repeat
  $\mathcal{OT} \leftarrow update(\mathcal{OT})$
  while $(\neg isClosed(\mathcal{OT}) \vee \neg isConsistent(\mathcal{OT}))$ **do**
    if $(\neg isClosed(\mathcal{OT}))$ **then**
      $\exists s_1 \in S, a \in A.\ \forall s \in S.\ row(s_1 \cdot a) \neq row(s)$
      $S \leftarrow S \cup \{s_1 \cdot a\}$
      $\mathcal{OT} \leftarrow update(\mathcal{OT})$
    end if
    if $(\neg isConsistent(\mathcal{OT}))$ **then**
      $\exists s_1, s_2 \in S, a \in A, e \in E.\ row(s_1) = row(s_2) \wedge T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$
      $E \leftarrow E \cup \{a \cdot e\}$
      $\mathcal{OT} \leftarrow update(\mathcal{OT})$
    end if
  end while
  $M_c \leftarrow M(\mathcal{OT})$
  $\sigma_c \leftarrow EO(M_c)$
  if $(\sigma_c \neq \perp)$ **then**
    $S \leftarrow S \cup Prefix(\sigma_c)$
  end if
until $(\sigma_c = \perp)$

MO (EO) denotes the call to the membership oracle (equivalence oracle) and the functions *update* resp. M are defined as follows:

— *update* : $\mathcal{OT} \rightarrow \mathcal{OT}$, where for each $s \in (S \cup S \cdot A)$ and $e \in E$, $T(s \cdot e) = MO(s \cdot e)$

— The *hypothesis* can be computed from the observation table as follows: $M : \mathcal{OT} \rightarrow \mathcal{DFA}$

- $\Sigma = \{row(s) | s \in S\}$,
- $\delta(row(s), a) = row(s \cdot a)$ $(a \in A)$,
- $s_0 = row(\lambda)$,
- $F = \{row(s) | s \in S \text{ and } T(s) = 1\}$.

(cf. Section 8.3.1). The problem of a practical implementation of a membership oracle resp. equivalence oracle is discussed in Section 8.3.2. Here particularly the problem of the connection to the "real" world is studied, which is realized by using the *ITE*. Afterwards we define the finite installations on which we have carried out our experiments to evaluate the learning efficiency (Section 8.3.3), before we illustrate the algorithm L^* along a concrete scenario (Section 8.3.4).

8.3.1 Formal Adaptations

To make L^* work in practice, there are several problems to solve. First of all, the worlds of automata learning and testing have to be matched. A reactive system cannot be readily seen as a finite automaton. It is a reactive, real-time system which receives and produces signals from large value domains. To arrive at a finite automaton, one has to abstract from timing issues, tags, identifiers and other data fields.

As we are dealing with reactive systems the formalism of *synchronous languages* like *Esterel* [BG92] or *Lustre* [HCP91] fits perfectly well. Reactive systems behave *deterministic*, i.e. the outputs of the system are uniquely determined by its inputs. But this is an important property for regression testing, because results need to be reproducible, i.e., results should not be subject to accidental changes (cf. Section 3.4). In the context of reactive systems, one observes that a system's reaction to a stimulus may consist of more than one output signal. We assume that there is no reordering between the outputs, i.e. they are produced everytime in a particular order. Those outputs are produced with some delay, and in everyday application some of them may actually occur only after the next outside stimulus has been received by the system. It is very important to be able to match outputs correctly to the stimuli which provoked them, i.e. to keep the causality. In synchronous languages this is conceptually ensured by assuming that the reactions take no time, or, equivalently that outputs become available as soon as inputs become available. In testing practice, however, it is common to wait after each stimulus to collect all outputs. Most often, appropriate timeouts are applied to ensure that the system has produced all responses and settled into a "stable" state. If all tests are conducted in this way, for interpreting, storing and comparing test results, a reactive system can be viewed as an *input/output deterministic finite automaton*: a device which reacts on inputs by producing outputs and possibly changing its internal state. It may also be assumed that the system is *input enabled*, i.e. that it accepts all inputs regardless of its internal state.

Furthermore, we would like to look at the system as a propositional, i.e. finite, object. This means that we would like to have only a finite number of components, input signals, and output signals. With respect to the number of components, again we are helped by common testing practice. The complexity of the testing task necessitates a restriction to small configurations. So only a small number of addresses and component identifications will occur, for instance up to four telephone devices in a CTI setting, and can accordingly be represented by propositional symbols. Furthermore, we abstract away other components of signals, e.g. like time stamps. This leads to deterministic systems responses to stimuli, which is another important prerequisite for reproducible, unambiguously interpretable test results. Finally, we will assume, that the system models we consider are all finite-state.

Summarizing, we adopted the following common assumptions and practices from real-life testing.

1. Distinction between stimuli and responses
2. Matching deterministic reactions to the stimuli that cause them (synchronous hypothesis)
3. System accepts all stimuli in every situation (input enabled)
4. Restriction to small installations
5. Abstraction to propositional signals
6. Abstract from timing issues (causality)

Altogether, this leads to a view of a reactive system as a propositional *Input/Output Deterministic Finite Automaton.*

Definition 8.4 (Input/Output Deterministic Finite Automaton). An input/output deterministic finite automata ($\mathcal{IODFA}$) is a structure $\mathcal{S} = (\Sigma, A_I, A_O, \delta, \chi, s_0)$, where

— Σ is a non-empty, finite set of states,

— A_I is the alphabet of input symbols,

— A_O is the alphabet of output symbols,

— $\delta : \Sigma \times A_I \to \Sigma$ is the transition function,

— $\chi : \Sigma \times A_I \to A_O^*$ is the output function,

— s_0 is the unique start state.

We write $s \xrightarrow{a/\sigma} s'$ if $\delta(s,a) = s'$ and $\chi(s,a) = \sigma$. Furthermore we will extend δ, χ to operate on strings, i.e. $\delta(s, a \cdot \sigma) = \delta(\delta(s,a), \sigma)$ and $\chi(s, a \cdot \sigma) = \chi(\delta(s,a), \sigma)$.

Input/output deterministic finite automata according to this definition are not to be confused with the richer structures from Definition 2.7 ([LT89]). As we are not concerned with automata algebra, we can use this simple form here[2].

Input/output deterministic finite automata differ from ordinary automata in that their edges are labelled with inputs and outputs instead of just one symbol. Obviously we could view a combination of an input and the corresponding output symbols as one single symbol. In our scenario this would result in a very large alphabet, particularly because of the sequences of elementary output symbols. Running L^* with such a large alphabet would be very inefficient. So we chose to split an edge labelled by a sequence of one input and several output symbols, i.e. $s \xrightarrow{a/\sigma} s'$, into a sequence of auxiliary states, connected by edges with exactly one symbol. In this way, we keep the alphabet and the observation table small (even though the number of states increases). Further reductions, which are possible because of the specific structure of the resulting automata, prohibit a negative impact of the state increase on the learning behaviour. They are discussed in Section 8.4.

Definition 8.5. The transformation of an input/output deterministic finite automaton $\mathcal{S} = (\Sigma^{io}, A_I, A_O, \delta^{io}, \chi^{io}, s_0^{io})$ into a $\mathcal{DFA}$ $DFA(\mathcal{S}) = (\Sigma, A, \delta, s_0, F)$, is given by:

— $A = A_I \cup A_O$,

— $\Sigma \supseteq \Sigma^{io} \cup \{s_\perp\}$, where $s_\perp$ is an artificial error state,

— $s_0 = s_0^{io}$,

— for each transition $\delta^{io}(s,a) = s'$ and $\chi(s,a) = a_1 \cdot \ldots \cdot a_n$, the set Σ contains *transient states* $s_1, \ldots s_n$, with transitions

  - $\delta(s,a) = s_1$ and $\delta(s_n, a_n) = s'$,
  - $1 \leq i < n.\ \delta(s_i, a_i) = s_{i+1}$ and $\delta(s_i, b) = s_\perp$ for each $b \in A \setminus \{a_i\}$

— transitions $\delta(s,b) = s_\perp$ for each $s \in \Sigma^{io}$ and $b \in A_O$,

— the transitions $\delta(s_\perp, a) = s_\perp$ for each $a \in A$,

— $F = \Sigma \setminus \{s_\perp\}$.

[2] Another common name for our type of machine is (deterministic) *Mealy Automaton*, only we permit a string of output symbols at each transition.

Given a system represented by an input/output deterministic finite automaton $\mathcal{S}$, our adaptation of L^* will learn a minimal $\mathcal{DFA}$ equivalent to $DFA(\mathcal{S})$. This requires the adequate realization of a membership and an equivalence oracle.

8.3.2 A Practical Implementation of a Membership and an Equivalence Oracle

In practice, membership tests may be available – and, in fact, in the application scenario considered here they are – but equivalence tests are out of reach, particulary in a black-box scenario. Nevertheless, L^* is useful in practice, as the (exact) equivalence oracle can be replaced by an approximative one. Key to the practical implementation of both oracles is, however, the definition of appropriate abstraction and concretization functions. Different from other situations where abstraction is applied, for instance validation by formal methods, it is of crucial importance here to be able to reverse the abstraction (concretization). Learning is only feasible if one can check actively whether a given abstract sequence corresponds to (is an abstraction of) a concrete system behaviour, and L^* relies heavily on being able to have such questions answered. I.e., in order to be able to resolve the *membership queries*, abstract sequences of symbols have to be retranslated into concrete stimuli sequences and fed to the system at hand.

Abstraction and Concretization

Membership queries can be answered by testing the system we want to learn. This is not quite as simple as it sounds, mainly because the sequence to be tested is an abstract, propositional string, and the system on the other hand is a physical entity whose interface follows a real-time protocol for the exchange of digital (non-propositional) data. Here the *ITE* helps bridging the gap between the abstract model and the concrete system. In principle we need to transform an abstract, propositional string into an *ITE test case* and the resulting *test run* back again into a propositional string.

The substitution of the parametric actions by propositional ones is to some extend comparable to the concept of *schematic names* of [JP89]. However, whereas schematic names are used because the programs are data-independent, i.e. their behaviours are independent of the actual data domain on which they operate, in our case some parameters are still important, while others can be simply disregarded. Thus it is still important to decide, which parameters are relevant and which not, e.g. internal timers can often be abstracted away, whereas concrete device identifiers are of crucial importance.

The key idea for the definition of the abstraction resp. concretization functions is given by the handling of parameterized actions by the model checker, cf. Section 3.4.3. There we also needed to transform parameterized actions into propositional ones. Thus we decided to encode the parameters and their values directly into the propositional action itself, in a style inspired by the parameter encoding in URL's. This enables us on the one hand to distinguish between actions with different parameter values. On the other hand we are able to define the concretization function easily. Note that these are pure syntactical transformations.

The following definition states the abstraction function f_A and its corresponding concretization function f_A^{-1}.

Definition 8.6. Let $\Pi = \{p_1, \ldots, p_n\}$ be the set of available formal test block parameters, $s \in \Sigma$ be a test block, and A_P be the set of propositional actions. Remember that $\mathcal{L}$ is the test block labelling function (cf. Definition 3.2), mapping each test block to a pair consisting of the identifier (id) of the test block (first component) and a partial parameter function (second component), which itself maps parameters to their values. Then the abstraction function f_A can be defined as follows:

$$f_A(s) \;=\; \underbrace{\overbrace{\mathcal{L}(s)\downarrow_1}^{id} \;?\; \overbrace{p_1 = \mathcal{L}(s)\downarrow_2(p_1)}^{p_1} \;?\; \ldots \;?\; \overbrace{p_n = \mathcal{L}(s)\downarrow_2(p_n)}^{p_n}}_{a \in A_P}$$

Furthermore the concretization function f_A^{-1} can be defined:

$$f_A^{-1}(id \;?\; p_1 = v_1 \;?\; \ldots \;?\; p_n = v_n) \;=\; \begin{array}{l} s, \text{ where } \mathcal{L}(s)\downarrow_1 = id, \text{ and} \\ \mathcal{L}(s)\downarrow_2(p_1) = v_1, \ldots, \mathcal{L}(s)\downarrow_2(p_n) = v_n \end{array}$$

Note that the set of input actions is determined by the corresponding set of external action test blocks.

Example 8.1. To illustrate the abstraction resp. concretization functions, let us consider a test block with the identifier *makeCall* which initiates a call between two telephone devices. These devices can be specified by means of two parameters, i.e. *source* and *target*. Let furthermore s denote an instance of this test block in a test graph where device A calls device B , i.e. $\mathcal{L}(s)\downarrow_1 = \mathit{makeCall}$, $\mathcal{L}(s)\downarrow_2(\mathit{source}) = A$, and $\mathcal{L}(s)\downarrow_2(\mathit{target}) = B$. Now

$$f_A(s) = \mathit{makeCall} \;?\; \mathit{source}{=}A \;?\; \mathit{target}{=}B$$

And on the other hand

$$f_A^{-1}(\; \underbrace{\mathit{makeCall}}_{\mathcal{L}(s)\downarrow_1} \;?\; \underbrace{\mathit{source}{=}A}_{\mathcal{L}(s)\downarrow_2(\mathit{source}) = A} \;?\; \underbrace{\mathit{target}{=}B}_{\mathcal{L}(s)\downarrow_2(\mathit{target}) = B} \;) = s$$

Membership Oracle

With these preconditions, the following steps have to be carried out to perform a membership query for a sequence σ:

(1) Compute the projection σ_I of σ to elements out of A_I only.

(2) Build a test case for σ_I. Here for each symbol a out of A_I the corresponding test block is computed by $f_A^{-1}(a)$. Note that the generation procedure for such test cases is straightforward, as they are composed out of action test blocks only, i.e. without branching. This is only possible because of our "loose" definition of test cases. In Section 3.4.1 we have established a test case specification formalism that allows us to focus on the relevant system behaviour, i.e. we do not have to check the complete system state after stimulations. In this situation we use this feature by checking nothing at all.

(3) Execute the test case. Due to the execution semantics of a test case, cf. Section 3.4.2, the resulting test run σ_r contains all corresponding responses of the system as well.

(4) Retransform the test run σ_r back into a propositional sequence σ_p by applying f_A to every element of σ_r. Note that eventually an output symbol, representing e.g. a structured protocol entity, must be transformed into the required notation of Definition 8.6, i.e. an identifier followed by key/value pairs for the mandatory parameters. During this step we additionally need to abstract from unnecessary elements, e.g. like timestamps or message identifiers.

(5) If the original sequence σ is included in the set of prefixes of σ_p ($\sigma \in prefix(\sigma_p)$), then the membership query can be answered with 1. Otherwise σ contains unexpected output symbols and the answer is 0.

In Figure 8.2 this process is illustrated for a concrete example. Let us assume that L^* raises a membership query for the sequence $a \cdot x \cdot b$ (a and b denote input symbols, whereas x and y are output symbols). Then during step (1) the projection to the input symbols will be performed, which results in $a \cdot b$. This sequence will then be transformed into a test case with the help of the concretization function f_A^{-1} (we have omitted the internal start test block to concentrate on the essentials of this test case only). The corresponding test graph, which represents its semantics, will be computed by the *ITE* test execution engine using the abstraction function f_A, cf. Section 3.4.2. Here the reflexive edges, labelled with the overall output alphabet ensure that all responses of the system that occur during test execution will be gathered during the test run (3). As the test run may contain unnecessary details

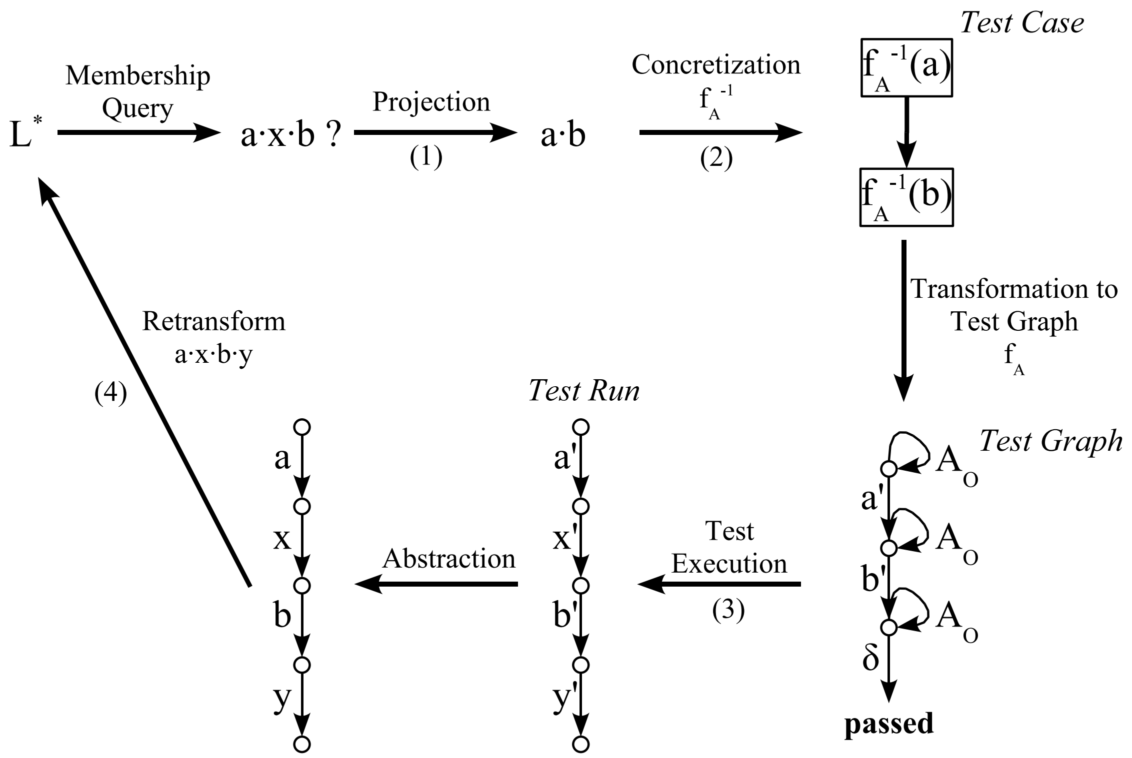


Figure 8.2: Illustration of a practical Membership Oracle

for the respective actions, e.g. timestamps or message identifiers for the responses of the system or test tool identifiers for the stimuli, we have to abstract from these details. During this abstraction step we simply disregard certain parameters that are encoded in the action (e.g. transformation of x' to x in Figure 8.2). Note that this does not conflict with the concretization function f_A^{-1}, as all parameters of a test block have predefined values so that every test block is always executable. Finally, we have to retransform the (abstract) test run back into a propositional sequence (4). The result for this particular membership query is then **true**, as the sequence $a \cdot x \cdot b$ is indeed a prefix of the result $a \cdot x \cdot b \cdot y$. If we have received e.g. the sequence $a \cdot y \cdot b \cdot x$ the result of the membership query would have been **false**.

Although the tests are automatically generated, membership queries are very expensive: The automatic execution of a single test took approximately 1,5 minutes during our experimentation. So it is very important to keep the number of such membership queries low.

Equivalence Oracle

For a black-box system there is obviously no reliable equivalence check – one can never be sure whether the whole behaviour spectrum of a system has been explored. But there are approximations which cover the majority of systems quite well. This is the basis of the theoretical result that approximate learning (learning with probably approximately correct results [Val84]) can be done with membership queries alone [KV94]. The basic idea is to scan the system in the vicinity of the explored part for discrepancies to the expected behaviour.

Substituting the equivalence queries of L^* turned out to be simple in our experiments. We used simple lookahead queries and were able to (manually) identify the structural properties of the learned models that we have expected. Assume that we have the membership query for input symbol $\sigma = \sigma_I = a$ which results in $\sigma_r = a \cdot x$, where x is an output symbol. For answering the membership question we simply have to compare σ and σ_r. The full information provided by the test run, i.e. that after an input action a an output action x occurred, can be stored as a new test in the equivalence oracle as well. This helps to guide the algorithm afterwards, as every hypothesis must also satisfy the system behaviour seen before.

Though we do not expect this to remain that easy if we move to more ambitious scenarios than the ones studied so far, approximate equivalence checks seem unlikely to pose difficulties.

One particularly good approximation is achieved by performing a test in the spirit of [TP98], cf. Section 2.3.4. This test procedure has polynomial complexity in the size of the hypothesis automaton. Note that for the real execution of the resulting test sequences, we can use the already implemented membership oracle. Another possibility lies in checking consistency within a fixed lookahead from all states. In the limit, by increasing the lookahead, this exploration will rule out any uncertainty, though at the price of exponentially increasing complexity [KV94].

Another possibility of enriching the source of information the equivalence oracle is based on the incorporation of expert knowledge. This has already been proposed in the general approach of regular extrapolation, cf. Section 8.1.1. On the one hand experts can specify special sequences that distinguish similar states (tests). On the other hand LTL constraints help to discover an incorrect hypothesis: if a constraint is violated, the model checker of the *ABC* is able to present a counterexample.

8.3.3 Scenarios for Experimentation

To illustrate our approach we have conducted several experiment, where we learned models for small installations of (a part of) a complex call center solution, cf. Section 6.2. In all of our scenarios, a telephone switch is connected to the telephone network and acts as a “normal” telephone switch to the phones. Additionally it communicates with CTI applications that are executed on PC’s. Here in essence two communication protocols are used: *CorNet* for the communication between the switch and its devices, and *CSTA/Tapi* for the communication between the switch and its CTI applications (cf. Figure 6.6). The technical realization of the necessary interface to this setup is provided by the *ITE*, i.e. we are particularly able to:

1. Stimulate the system by means of the *CorNet* protocol, and
2. Observe the system with respect to the resulting CSTA messages.

A model on CSTA level of the telephone switch will be of practical interest, because in Section 6.1 it was stated that there exist more than 200 different CTI applications and each communicates via the CSTA protocol with the telephone switch. As the telephone switch is involved into all of these settings, a model of the behaviour of the telephone switch on CSTA level helps to steer the testing effort.

We have carried out our experiments on four specific (partial) installations of a call center solution, each consisting of the telephone switch connected to a fixed number of telephones (called “physical devices”)[3]. Our main aims are to present a qualitative evaluation of our approach and to study the effects of independency between actions. Therefore the telephones were restricted differently in their behaviour, ranging from simple on-hook (↑) and off-hook (↓) actions of the receiver to actually performing calls (→). The four installations are listed below. Note that we have two versions of the first three scenarios, depending on the observability of the signal (*hookswitch*).

In the following we present a short description of our experimental settings together with the corresponding input resp. output alphabet.

Scenario 1 (S_1): One telephone device is connected to the telephone switch and it is able to on-hook resp. off-hook the receiver.

1 physical device (A),
$A_I = \{A\uparrow, A\downarrow\}$,
$A_O = \{init_A, clear_A, [hookswitch_A]\}$.

Scenario 2 (S_2): Two telephone devices are involved in this setting and each is able to on-hook resp. off-hook its receiver. The actions depending on device A are independent from the actions concerning device B.

2 physical devices (A, B),

[3]Note that this restriction is sufficient for the investigation of the relevant behaviour of the CSTA protocol [Eur00].

$A_I = \{A\uparrow, A\downarrow, B\uparrow, B\downarrow\}$,
$A_O = \{init_{\{A,B\}}, clear_{\{A,B\}}, [hookswitch_{\{A,B\}}]\}$.

Scenario 3 (S_3): Three independent telephone devices are included in this scenario.

3 physical devices (A, B, C),
$A_I = \{A\uparrow, A\downarrow, B\uparrow, B\downarrow, C\uparrow, C\downarrow,\}$,
$A_O = \{init_{\{A,B,C\}}, clear_{\{A,B,C\}}, [hookswitch_{\{A,B,C\}}]\}$.

Scenario 4 (S_4): The last setting incorporates three telephone devices and device A can call device B, i.e. they are eventually interdependencies between these devices.

3 physical devices (A, B, C),
$A_I = \{A\uparrow, A\downarrow, A\rightarrow B, B\uparrow, B\downarrow, C\uparrow, C\downarrow,\}$,
$A_O = \{init_{\{A,B,C\}}, clear_{\{A,B,C\}}, orig_{A\rightarrow B}, estbl_B\}$

The output events indicate which actions the telephone switch has performed on the particular input. For instance, $init_A$ indicates that the switch has noticed that the receiver of device A is off the hook, whereas $init_{\{A,B\}}$ abbreviates the situation that we have both events $init_A$ and $init_B$. With $orig_{A\rightarrow B}$ the switch reports that it has forwarded the call request from A to B.

8.3.4 Basic Learning in Practice

Let us discuss the practical application of the basic learning algorithm L^* by computing the scenario S_2.

Example 8.2. Let us assume that the alphabet is given by $A = A_I \cup A_O = \{A\uparrow, B\uparrow, A\downarrow, B\downarrow, init_A, init_B, clear_A, clear_B\}$. Then the initial observation table is given by (1).

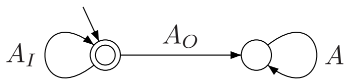
	λ
λ	1
$A \uparrow$	1
$B \uparrow$	1
$A \downarrow$	1
$B \downarrow$	1
$\boxed{init_A}$	0
$init_B$	0
$clear_A$	0
$clear_B$	0

(1) Initial Observation Table

	λ
λ	1
$init_A$	0
$A \uparrow$	1
$B \uparrow$	1
$A \downarrow$	1
$B \downarrow$	1
$init_A \cdot A^*$	0
$init_B$	0
$clear_A$	0
$clear_B$	0

(2) Closed and Consistent Observation Table

The initial observation table (1) is not closed, because there are rows out of $S \cdot A$, e.g. $row(init_A)$, with no corresponding row in the set S, i.e. $\forall s \in S.row(s) \neq row(init_A)$. Consequently the string of one of these rows, e.g. $init_A$, will be added to the set S. Afterwards the observation table will be updated, i.e. for each s out of $(S \cup S \cdot A)$ and e out of E the corresponding value for $T(s \cdot e)$ will be updated by membership queries. The right observation table[4] (2) is then closed and consistent and the corresponding hypothesis will be computed:



The equivalence oracle rejects the hypothesis, because e.g. after an off-hook for device A the telephone switch responds with the signal $init_A$. Thus $\sigma_c = A \uparrow; init_A \in \mathcal{L}$ is not reflected in the hypothesis. Note that we have added the (possible) counterexample σ_c to the equivalence oracle when performing the membership query for $A \uparrow$. The counterexample σ_c, and all its prefixes, will be added to set S, which results in the following observation table (3).

[4]Note that we present the observation table in a somewhat compressed view, as e.g. entry $init_A \cdot A^*$ denotes a set of entries, e.g. $init_A \cdot A \uparrow$, $init_A \cdot B \uparrow$ and so forth.

	λ
λ	1
$init_A$	0
$A\uparrow$	1
$A\uparrow \cdot init_A$	1
$B\uparrow$	1
$A\downarrow$	1
$B\downarrow$	1
$init_A \cdot A^*$	0
$init_B$	0
$clear_A$	0
$clear_B$	0
$A\uparrow \cdot A_I^*$	0
$A\uparrow \cdot (A_O \setminus \{init_A\})^*$	0
$A\uparrow \cdot init_A \cdot A_I$	1
$A\uparrow \cdot init_A \cdot A_O^*$	0

(3) Inconsistent Observation Table

	λ	$init_A$
λ	1	0
$init_A$	0	0
$A\uparrow$	1	1
$A\uparrow \cdot init_A$	1	0
$B\uparrow$	1	0
$A\downarrow$	1	0
$B\downarrow$	1	0
$init_A \cdot A^*$	0	0
$init_B$	0	0
$clear_A$	0	0
$clear_B$	0	0
$A\uparrow \cdot A_I^*$	0	0
$A\uparrow \cdot (A_O \setminus \{init_A\})^*$	0	0
$A\uparrow \cdot init_A \cdot A_I$	1	0
$A\uparrow \cdot init_A \cdot A_O^*$	0	0

(4)

The left observation table (3) is not consistent, because $row(\lambda) = row(A\uparrow)$ but $row(init_A) \neq row(A\uparrow; init_A)$. Thus the string $\lambda{\cdot}init_A$ will be added to set E and the observation table will be updated accordingly. Note that the resulting observation table (4) is still not consistent ($row(\lambda)$ and $row(A\uparrow \cdot init_A)$). We will now suppress the intermediate steps and present directly the final hypothesis in Figure 8.3. Note that in the hypothesis all disregarded actions lead to the non-accepting state $\perp$, but are not depicted explicitly.

Results

Table 8.1: Number of Membership Queries for Basic Learning

Scenario	States	Membership Queries
S_1	5	108
S_1'	7	672
S_2	13	2,431
S_2'	29	15,425
S_3	33	19,426
S_3'	81	132,340
S_4	79	132,300

Table 8.1 presents the sizes of the learned automata and the number of membership queries needed for the learning procedure. The third column of Table 8.1 points out impressively that even in relatively small scenarios the number of membership

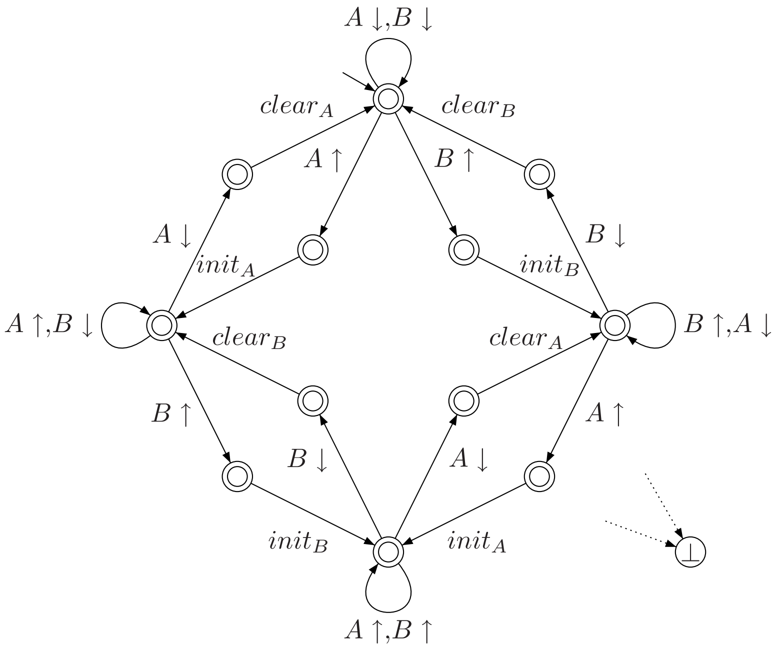


Figure 8.3: Final Model for Scenario S_2

queries needed is quite large. Therefore it is clearly necessary to reduce the number of membership queries drastically. In fact, we would have had difficulties in computing these figures in time without using some reductions[5]. The next section formally develops this novel application of a profile-dependent optimization of the learning process, which we consider a key for the practicality of automata learning.

8.4 Optimized Learning with L^*

A major cause of the difficulties is the fact that L^* – as all its companions – does not take peculiarities of the application domain into account. Though a suitable abstraction of a telecommunication system does indeed yield a finite automaton, it is a very special one: Its set of finite traces forms a prefix-closed language, every other symbol in the trace is an output determined by the inputs received by the system before, and, assuming correct behaviour, the trace set is closed under rearrangements of independent events. It was this last property which caused the most problems when adapting the learning scenario. In fact, identifying independence and the concrete corresponding exploitation for the reduction of memberships queries both

[5]We have learned the corresponding automata directly with our optimized version of L^*, which we will present in the next section. However, to evaluate the impact of our optimizations we have used theses models to trigger the basic algorithm L^*.

turned out to be difficult. However, since physical testing, even if automated as in our case, is very time intensive, these optimizations are vital for practicality reasons.

To reduce the huge amount of membership queries needed in the case of basic L^*, we propose a process which is presented in Fig. 8.4. Here, additional filters are added to the connection from L^* to the two oracles. These filters reduce the number of queries to the oracles by answering queries themselves in cases where the answer can be deduced from previous queries. In our experiments, we used properties like *determinism*, *prefix closure*, and also *independence* and *symmetry* of events for filter construction.

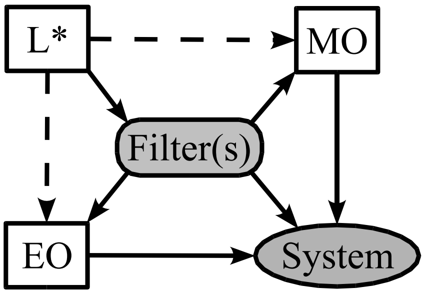


Figure 8.4: L^* with oracle substitutes

8.4.1 Theory

This section describes several optimizations given through rules which can be used in the filter shown in Figure 8.4. Common to all rules is that they are able to answer the membership queries from within the accumulated knowledge of the system so far. They will be presented in the form `Condition` $\Rightarrow$ $\{$`true`, `false`$\}$. `Condition` refers to the $\mathcal{OT}$, in particular to T, to find out whether there is an entry in the table stating that some string σ is already known to be member of the set to be learned or not, i.e. $T(\sigma)$.

The filter rules that will be presented are based on several properties of (special classes of) reactive systems. They are to some extent orthogonal to each other and could be applied independently. This potentially increases the usability both of our current concrete approach and of the general scenario depicted in Fig. 8.4. Given another learning problem, a possible way to adapt the learning procedure is to identify a profile of the new scenario in terms of specific properties and to optimize the learning process by means of the corresponding filter rules.

Prefix-Closure:

In testing a reactive system, one observes the finite prefixes of the potentially infinite sequences of inputs and outputs. If there is an error, all further behaviour is disregarded. Other than in general regular languages, there is no switching from non-accepting to accepting. Correspondingly, the DFA's resulting from translating an input/output deterministic finite automaton according to Definition 8.5 have just one sink as a non-accepting state. The language we want to learn is thus *prefix closed*.

Definition 8.7 (Prefix Closure). A Language $\mathcal{L}$ over an alphabet A is called prefix closed, if the following holds:

— If $\sigma = \sigma' \cdot \sigma''$ then σ' is called a *prefix* of σ. The set $prefix(\sigma)$ denotes all prefixes of σ.

— $\forall \sigma \in A^*.\ \sigma \in \mathcal{L} \ \Rightarrow \forall \sigma' \in prefix(\sigma).\ \sigma' \in \mathcal{L}$.

This property directly gives rise to the rule Filter 1, where each prefix of an entry in $\mathcal{OT}$ that has already been rated 1 (i.e. is a member of the target language), will be evaluated with `true` as well. Note that the filter rules are valid for all sequences σ.

Filter 1 (Positive Prefix). $\exists \sigma' \in A^*.\ T(\sigma \cdot \sigma') = 1 \Longrightarrow MO(\sigma) = true$

The contraposition of the condition in Definition 8.7 states that once we have found a sequence σ that is rated `false`, all continuations also have to be rated `false`. This yields the rule Filter 2.

Filter 2 (Negative Prefix). $\exists \sigma' \in prefix(\sigma).\ T(\sigma') = 0 \Longrightarrow MO(\sigma){=}false$

Together, these two rules capture all instances of membership queries which are avoidable when learning prefix-closed languages. These rather simple rules already have an impressive impact on the number of membership queries as will be shown in Table 8.2.

Input Determinism:

Input determinism ensures that the system to be tested always produces the same outputs on any given sequence of inputs (cf. Definition 2.7). This implies that replacing just one output symbol in a word of an input deterministic language cannot lead to words of this same language.

Proposition 8.8. Let $\mathcal{S}$ be an input/output deterministic finite automaton and let furthermore $\sigma \in \mathcal{L}(DFA(\mathcal{S}))$. Then the following holds.

1. If there exists a decomposition of $\sigma = \sigma' \cdot x \cdot \sigma''$ with $x \in A_O$, then $\forall y \in A \setminus \{x\}.\ \sigma' \cdot y \cdot \sigma''' \notin \mathcal{L}(\mathcal{S})$.
2. If there exists a decomposition of $\sigma = \sigma' \cdot a \cdot \sigma''$ with $a \in A_I$, then $\forall x \in A_O.\ \sigma' \cdot x \cdot \sigma''' \notin \mathcal{L}(\mathcal{S})$.

The first property says that each single output event is determined by the previous inputs. It should be emphasized that this property is of crucial importance for learning reactive systems, as a test environment has no direct control over the outputs of a system. If the outputs were not determined by the inputs, there would be no way to steer the learning procedure to exhaustively explore the system under consideration.

The second property reflects the fact that the number of output events in a given situation is determined, and that we wait with the next stimulus until the system has produced all its responses. This is useful but not as indispensable as the first property. The corresponding filter rules are straightforward:

Filter 3 (Input Determinism). $\exists x \in A_O, y \in A, \sigma', \sigma'', \sigma''' \in A^*.\sigma = \sigma' \cdot x \cdot \sigma'' \wedge T(\sigma' \cdot y \cdot \sigma''') = 1 \wedge x \neq y \implies MO(\sigma) = false$

Filter 4 (Output Completion). $\exists a \in A_I, x \in A_O, \sigma' \in A^*.\sigma' \cdot x \in prefix(\sigma) \wedge T(\sigma' \cdot a) = 1 \implies MO(\sigma) = false$

Independence of Events:

Reactive systems exhibit very often a high degree of parallelism. *Partial order reduction* methods for communicating processes [Maz87, Val93] deal with this kind of phenomena. Normally these methods can help to avoid examining all possible interleavings among processes. However, as we will illustrate here, these methods can also be used to avoid unnecessary membership queries using the following intuition of *independency*: If device A can perform a certain action and device B can perform another and they are independent of each other, then they can perform it vice versa as well.

Following the work of Mazurkiewicz [Maz87], we define a *trace equivalence* based on the notation of *independent actions*.

Definition 8.9 (Dependency, Independency). A relation $D \subseteq A \times A$ on a set A that is finite, reflexive, and symmetric is called *dependency*. Given dependency D the relation $I_D = (A \times A) \setminus D$ is called *independency* induced by D.

Definition 8.10 (Trace Equivalence). Let D be a dependency relation and A be the corresponding alphabet. Let furthermore A^* represent the set of all finite sequences of symbols in A. We define the relation $\equiv_{PO}$ as the least congruence in the monoid $[A^*, \cdot, \lambda]$ such that

$$(a, b) \in I_D \implies a \cdot b \equiv_{PO} b \cdot a$$

The relation $\equiv_{PO}$ is referred to as the *trace equivalence* over D and A. We use $[\sigma]_{PO}$ to denote the equivalence class of all traces equivalent to σ.

In the trace semantics of Mazurkiewicz, the behaviour of systems is defined as a set of traces. This semantics is often referred to as being a *partial order semantics* because it is possible to define a correspondence between traces and partial orders of occurrences of transitions [Maz87].

The membership question for a sequence σ can be improved, by answering it with `true` if an equivalent sequence σ' is already rated with 1 in $\mathcal{OT}$. Thus we obtain the following filter rule.

Filter 5 (Partial Order). $\exists \sigma' \in [\sigma]_{PO}.T(\sigma') = 1 \implies MO(\sigma) = true$

Whereas the general principle of Filter 5 is quite generic, the realization of the computation of the equivalent sequences strongly depends on the concrete application domain. However, the required process always follows the same pattern:

1. An expert specifies an application-specific **independence relation**, e.g. *Two independent calls can be shuffled in any order.*
2. σ is inspected to discover independent subparts with respect to the independence relation.
3. Computation of the partial order completion, i.e. the set of all equivalent traces.

In the following we will illustrate how the equivalent sequences for σ can be determined in the context of telephony systems by computing a simple example.

Example 8.3. Figure 8.5 shows the process of how to generalize a trace σ[6].

Device A calls device B and after that device B answers the call so that it will be established. As a last action, device C also picks up the receiver. However, this

[6]Note that input actions are prefixed by a question mark and output actions by an exclamation mark.

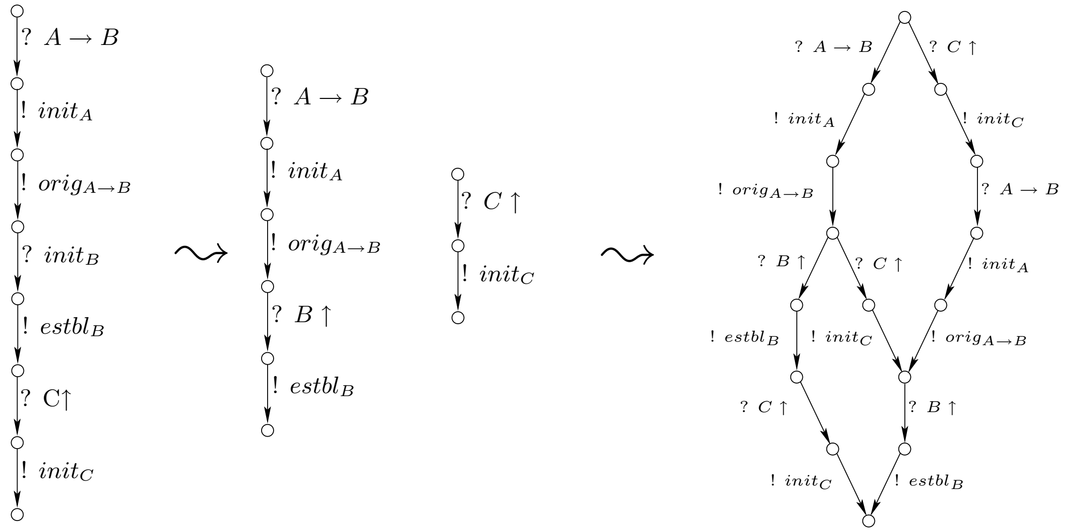


Figure 8.5: Partial Order Generalization

very last action is independent of the previous ones, as device C is not involved in the connection between A and B. Therefore, it does not matter, at which time C performs its action. Note that due to the strong coherence between the inputs and their corresponding outputs we combine inputs and outputs to new symbols for the computation of the interleavings. So the corresponding dependency of the example in Figure 8.5 is as follows:

$$D = \{(A \to B \cdot init_A \cdot orig_{A\to B}, B\uparrow \cdot estbl_B)\}$$

To compute all equivalent sequences we first will compute the dependencies between the involved devices, which leads to two independent subsequences. Following this, from these two independent subsequences, which describe the partial order, a directed acyclic graph representing all possible interleavings is computed and this represents the set of all equivalent traces.

Symmetries of Events:

Another observation in telecommunication systems is that several different components of one type, here e.g. telephones, can often be regarded as generic so that they are interchangeable. This leads to *symmetries*, e.g. a device A behaves like device B. Again this helps to avoid unnecessary membership queries: If we have seen that device A can perform a certain action we know that the other (similar) devices can perform it as well. Note that this effect is often related to the independence of events, within this section, however, we discuss these effects separately.

To compute the set of symmetric traces for a sequence σ we first compute an *abstract trace*, where we replace all concrete device identifiers with abstract actors. Afterwards we concretize the abstract trace again, i.e. we assign all possible values to the abstract actors. This way we are able to compute the set of symmetric traces.

Definition 8.11 (Abstract Trace, Symmetry). Let $Dev = \{A_1, \ldots, A_n\}$ be a set of different devices and $Act = \{\alpha_1, \ldots, \alpha_n\}$ be a set of actors.

1. The *abstract trace* for a sequence σ can be computed with the following algorithm:

   Initialize *actorDeps* with $\emptyset$, σ_α with λ, and *maxActor* with 0
   while $(\sigma \neq \lambda)$ **do**
     Decompose $\sigma = a \cdot \sigma'$
     Compute the device dependencies for a, i.e. $\{A_i, \ldots, A_j\}$
     for all $(A \in \{A_i, \ldots, A_j\})$ **do**
       if $(A \notin \mathit{actorDeps})$ **then**
         Increase *maxActor*
         Add $(A, \alpha_{maxActor})$ to *actorDeps*
       end if
     end for
     Replace the device id's in a with their corresponding actor id's
     Append a to σ_α
     $\sigma \leftarrow \sigma'$
   end while

2. The set of all *symmetric traces* for a sequence σ, denoted by $[\sigma]_S$, can can be computed by replacing the actors in σ_α with all possible concrete devices, i.e. $\alpha_1 = A_1, \alpha_2 = A_2, \ldots$ and $\alpha_1 = A_2, \alpha_2 = A_1, \ldots$ and so forth.

The corresponding filter rule is quite similar to Filter 5, as again the membership query for σ can be answered with `true` if a symmetric sequence σ' is already rated with 1 in $\mathcal{OT}$.

Filter 6 (Symmetry). $\exists \sigma' \in [\sigma]_S . T(\sigma') = 1 \implies MO(\sigma) = \mathit{true}$

Taken together these two filters provide powerful optimizations for the learning setting, cf. Section 8.4.2.

Please note that the two presented properties of telecommunication systems, i.e. independency and symmetries of events, can help to improve the approximate equivalence oracle as well. For every trace, all equivalent traces can be added to the equivalence oracle, as they are valid traces for the system as well. This means that a correct hypothesis must also accept these traces!

8.4.2 Optimized Learning in Practice

To understand how our optimizations work in practice let us investigate how the observation table (3) of Example 8.2 can be built with the help of the filters.

Example 8.4. In the following observation table each entry, that requires a "real" membership query is boxed in. Additional information about the query, e.g. the filter used to deduce the entry or all equivalent traces, with respect to the partial order, are presented in the last column.

	λ	Remarks
$\boxed{\lambda}$	1	λ
$init_A$	0	Filter 4
$\boxed{A\uparrow}$	1	$A\uparrow \cdot init_A$, $B\uparrow \cdot init_B$
$A\uparrow \cdot init_A$	1	Already seen
$B\uparrow$	1	Filter 5
$\boxed{A\downarrow}$	1	$A\downarrow$, $B\downarrow$
$B\downarrow$	1	Filter 5
$init_A \cdot A^*$	0	Filter 2
$init_B$	0	Filter 4
$clear_A$	0	Filter 4
$clear_B$	0	Filter 4
$A\uparrow \cdot A_I^*$	0	Filter 2 + 4
$A\uparrow \cdot (A_O \setminus \{init_A\})^*$	0	Filter 2, 3 + 4
$\boxed{A\uparrow \cdot init_A \cdot A\uparrow}$	1	$A\uparrow \cdot init_A \cdot A\uparrow$, $B\uparrow \cdot init_B \cdot B\uparrow$
$\boxed{A\uparrow \cdot init_A \cdot B\uparrow}$	1	$A\uparrow \cdot init_A \cdot B\uparrow \cdot init_B$, $B\uparrow \cdot init_B \cdot A\uparrow \cdot init_A$
$\boxed{A\uparrow \cdot init_A \cdot A\downarrow}$	1	$A\uparrow \cdot init_A \cdot A\downarrow \cdot clear_A$, $B\uparrow \cdot init_B \cdot B\downarrow \cdot clear_B$
$\boxed{A\uparrow \cdot init_A \cdot B\downarrow}$	1	$A\uparrow \cdot init_A \cdot B\downarrow$, $B\uparrow \cdot init_B \cdot A\downarrow$, $B\downarrow \cdot A\uparrow \cdot init_A$, $A\downarrow \cdot B\uparrow \cdot init_B$
$A\uparrow \cdot init_A \cdot A_O^*$	0	Filter 2 + 4

Let's discuss in detail how the result of a membership query can be induced by applying the filter rules.

— The sequence $init_A$ can be rated with 0 because from the membership queries for λ and $A\uparrow$ we can conclude with Filter 4: $\sigma = init_A, x = init_A$, $\sigma' = \lambda, a = A\uparrow$ and because $T(A\uparrow) = 1$ the query will be answered with $false$. Furthermore any continuation of $init_A$ can be rated with 0 because of Filter 2 (row $init_A \cdot A^*$).

— The sequence $B\uparrow$ can be answered because we have already queried the equivalent row $A\uparrow$ (Filter 5).

— The sequence $A \uparrow \cdot (A_O \setminus \{init_A\})$ can be rated with 0 because of Filter 3: e.g. $\sigma = A \uparrow \cdot init_B, \sigma' = A \uparrow, x = init_B, \sigma'' = \lambda$ but $T(A \uparrow \cdot init_A) = 1$. Furthermore because of Filter 2 + 3 we can also conclude that indeed every $A \uparrow \cdot (A_O \setminus \{init_A\})^*$ will be rated with 0.

Results

In the following, we discuss the impact of the filters on the scenarios defined in Section 8.3.3. Note that our filters do not affect the algorithm L^* itself and therefore none of its properties (correctness and termination). We have run the learning process with all available filters applied in a cumulative way, i.e., when using filters 3 and 4, filters 1 and 2 were also used. The results are summarized in Table 8.2, and Figure 8.6 visualizes the results on a logarithmic scale. The factor columns in the table provide the additional reduction factor in the number of membership queries achieved by successively adding new filters.

Table 8.2: Number of Membership Queries

Scen.	no Fil.	Fil. 1 & 2	Fact.	Fil. 3 & 4	Fact.	Fil. 5 & 6	Fact.	Tot.
S_1	108	30	3.6	15	2.0	15	0.0	7.2
S_1'	672	181	3.7	31	5.8	31	0.0	21.7
S_2	2,431	593	4.1	218	2.7	98	2.2	24.8
S_2'	15,425	4,080	3.8	355	11.5	145	2.4	106.4
S_3	19,426	4,891	4.0	1,217	4.0	207	5.9	93.8
S_3'	132,340	36,374	3.6	2,007	18.1	289	6.9	458.0
S_4	132,300	29,307	4.5	3,851	7.6	1,607	2.4	82.3

As one can see we were able to reduce the total number of membership queries in all scenarios drastically. In the most drastic case (S_3'), we only needed a fraction of a percent of the number of membership queries that the basic L^* would have needed. In fact, learning the corresponding automaton without any optimizations would have taken about 4.5 months of computation time.

The prefix reduction (filters 1 and 2) has a similar impact on all considered scenarios. This seems to indicate that it does not depend very much on the nature of the example and on its number of states.

The other two reductions (input determinism and partial order) vary much more in their effectiveness. Note that the saving factor increases with the number of states. Shifting attention to the number of outputs and the lengths of output sequences between inputs, these seem to have a particularly high impact on the effects of the filters 3 and 4. This can be seen by comparing the scenarios S_i with their counterparts S_i'. In these counterparts an additional output event is modelled, the hookswitch event, which occurs very frequently, namely after each of the permitted inputs.

One naturally expects that the impact of Filter 5, which covers the partial order aspects, will increase with the level of independency. And indeed we find this hypothesis confirmed by the experiments. S_1 has only one actor so that there is no independence, which results in a factor of 1. As the number of independent devices increases, the saving factor increases as well, see the figures for S_2 and S_3. It is worth noting that the number of states does not seem to have any noticeable impact on the effectiveness of this filter, as it is more or less remains constant when switching from S_i and S_i'. Compared to S_3, the saving factor in S_4 decreases. This is due to the fact that an action has been added in S_4 that can establish dependencies between two devices, namely the initiation of a call.

We have demonstrated that with the right modifications to the learning procedure, and the right environment for observing system, it is possible to learn system models. Modifying the learning algorithm by filtering out unnecessary queries enabled us to perform the experiments easily, and in general the method provides a flexible approach which permits fast adaptations to different application domains.

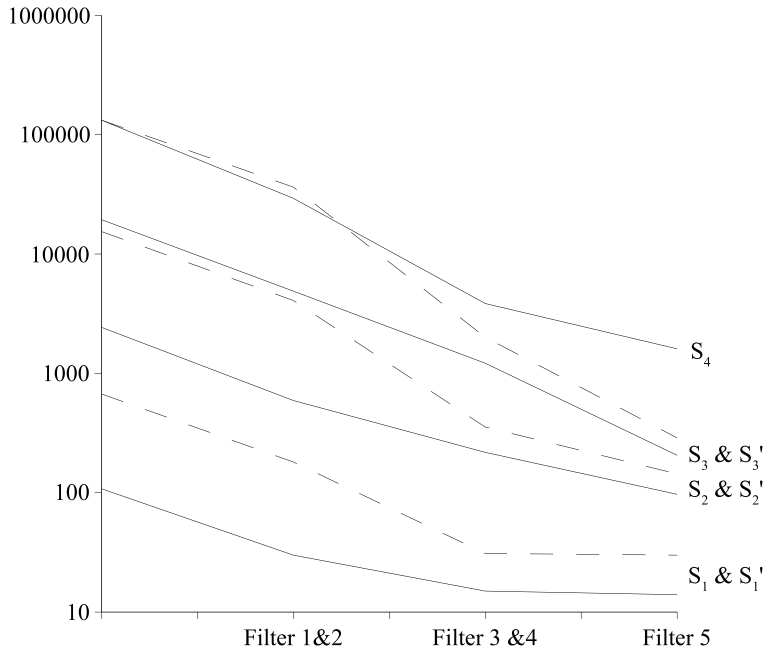


Figure 8.6: Impact of the filter rules on the number of membership queries (logarithmic scale)

8.5 $L^*_{i/o}$: Learning Input/Output Deterministic Systems

For fixed settings, modifications to the algorithm itself should be made, because this allows us to cope better with the peculiarities of special sorts of systems, i.e. input/output deterministic finite systems. When learning these systems directly as input/output deterministic finite automata, the overall performance of the learning algorithm can be further improved. This is mainly because less states are needed to represent such systems as $\mathcal{IODFA}$ in comparison to $\mathcal{DFA}$, so that no intermediate states are needed anymore.

When dealing with input/output deterministic finite systems the distinction between accepting and non-accepting states, as used in L^*, is not applicable. This is because the corresponding system model, $\mathcal{IODFA}$, only consists of "normal" states. Therefore, in testing theory for finite state machines or mealy automata, e.g. [Vas73, Cho78], states are distinguished with respect to the outputs they produce after applying (specific) inputs or sequences of those. We can generalize the observation table of [Ang87] (cf. Definition 8.2), so that the column/row entries consists of sequences of input symbols and the cell entries of sequences of output symbols.

Definition 8.12 (Input/Output Observation Table). An observation table for an input alphabet A_I is a triple $\mathcal{OT}_{i/o} = (S, E, T_{i/o})$, where $S \subset A_I^*$ is a finite, prefix-closed set, $E \subset A_I^*$ is a finite set, and $T_{i/o}$ is a finite function mapping strings of $(S \cup S \cdot A_I) \cdot E$ to strings from an output alphabet A_O. Furthermore if s is an element of $(S \cup S \cdot A_I)$, then $row_{i/o}(s)$ denotes the finite function f from E to A_O^* defined by $f(e) = T_{i/o}(s \cdot e)$.

1. $\mathcal{OT}$ is called **closed**, if $\forall t \in S \cdot A_I.\ \exists s \in S.\ row_{i/o}(t) = row_{i/o}(s)$.
2. $\mathcal{OT}$ is called **consistent**, if $\forall s_1, s_2 \in S.\ row_{i/o}(s_1) = row_{i/o}(s_2) \Rightarrow$ $\forall a \in A_I.\ row_{i/o}(s_1 \cdot a) = row_{i/o}(s_2 \cdot a)$.

Note that in the following we refer to an input/output observation table simply as observation table.

So the result of $T_{i/o}(s \cdot e)$, where $s \in (S \cup S \cdot A_I)$ and $e \in E$, will be exactly the sequence of output symbols that are produced after the sequence of input symbols $s \cdot e$ is executed. Additionally the set E of distinguishing sequences will be initialized with the whole set of input symbols A_I, because initially we try to distinguish between the states by applying every symbol of A_I.

To fill the entries in the observation table a membership oracle is needed that returns the (last) string of output symbols, which is produced after a sequence of input symbols. Note that this information is already available in our implementation of the membership oracle as defined for L^*, cf. Figure 8.2. There we have checked whether the result of the processed test run (σ_p) is in the set of prefixes of the original sequence σ. Now we will request membership queries for sequences composed out of input symbols only, i.e. the projection to input symbols is superfluous. The result of the membership query is then the suffix of σ_p, starting at the last input symbol.

The algorithm $L^*_{i/o}$ is given in Algorithm 8.2. Note that because $L^*_{i/o}$ learns the models directly as $\mathcal{IODFA}$ it subsumes the optimizations regarding prefix-closeness and input determinism for obvious reasons. For further improvements we will still use the partial order filter (Filter 5) and the symmetry filter (Filter 6).

8.5.1 Input/Output Learning in Practice

In the following example we will illustrate the power of $L^*_{i/o}$ along the scenario S_2 of Example 8.2.

Example 8.5. The alphabet is given by $A_I = \{A\uparrow, B\uparrow, A\downarrow, B\downarrow\}$ and $A_O = \{init_A, init_B, clear_A, clear_B\}$. The initial observation table is as follows:

	$A\uparrow$	$B\uparrow$	$A\downarrow$	$B\downarrow$
λ	$init_A$	$init_B$	λ	λ
$\boxed{A\uparrow}$	λ	$init_B$	$clear_A$	λ
$\boxed{B\uparrow}$	$init_A$	λ	λ	$clear_B$
$A\downarrow$	$init_A$	$init_B$	λ	λ
$B\downarrow$	$init_A$	$init_B$	λ	λ

Let us briefly discuss how the entries in $\mathcal{OT}$ will be interpreted. E.g. $T_{i/o}(A\uparrow \cdot B\uparrow) = init_B$ comes from the membership query for $A\uparrow \cdot B\uparrow$ which leads to the test run $\sigma_r = A\uparrow \cdot init_A \cdot B\uparrow \cdot \boxed{init_B}$. The result given back from the membership oracle will be exactly the boxed part of σ_r, i.e. $init_B$. If we are considering the entry $T_{i/o}(A\uparrow \cdot B\downarrow) = \lambda$ we see that the membership query for $A\uparrow \cdot B\downarrow$ results in $\sigma_r = A\uparrow \cdot init_A \cdot B\downarrow$. So the last input action results in no corresponding output action, i.e. the empty sequence λ.

This observation table is not closed, as for the two boxed rows, which are different from each other, no corresponding row in S exists. Thus we will add the two target rows to S which leads to the following observation table.

Algorithm 8.2: $L^*_{i/o}$

Input Alphabet A_I, Output Alphabet A_O, Observation Table $\mathcal{OT}_{i/o} = (S, E, T_{i/o})$, where initially $S = \{\lambda\}$ and $E = A_I$.

repeat
  $\mathcal{OT}_{i/o} \leftarrow update_{i/o}(\mathcal{OT}_{i/o})$
  while $(\neg isClosed(\mathcal{OT}_{i/o}) \vee \neg isConsistent(\mathcal{OT}_{i/o}))$ **do**
    if $(\neg isClosed(\mathcal{OT}_{i/o}))$ **then**
      $\exists s_1 \in S, a \in A_I.\ \forall s \in S.\ row_{i/o}(s_1 \cdot a) \neq row_{i/o}(s)$
      $S \leftarrow S \cup \{s_1 \cdot a\}$
      $\mathcal{OT}_{i/o} \leftarrow update_{i/o}(\mathcal{OT}_{i/o})$
    end if
    if $(\neg isConsistent(\mathcal{OT}_{i/o}))$ **then**
      $\exists s_1, s_2 \in S, a \in A_I, e \in E.\quad row_{i/o}(s_1) = row_{i/o}(s_2) \wedge$
      $\qquad T_{i/o}(s_1 \cdot a \cdot e) \neq T_{i/o}(s_2 \cdot a \cdot e)$
      $E \leftarrow E \cup \{a \cdot e\}$
      $\mathcal{OT}_{i/o} \leftarrow update_{i/o}(\mathcal{OT}_{i/o})$
    end if
  end while
  $M_c \leftarrow M(\mathcal{OT}_{i/o})$
  $\sigma_c \leftarrow EO_{i/o}(M_c)$
  if $(\sigma_c \neq \perp)$ **then**
    $S \leftarrow S \cup Prefix(\sigma_c)$
  end if
until $(\sigma_c = \perp)$

$MO_{i/o}$ ($EO_{i/o}$) denotes the call to the membership oracle (equivalence oracle) and the functions $update_{i/o}$ resp. M are defined as follows:

— $update_{i/o} : \mathcal{OT}_{i/o} \rightarrow \mathcal{OT}_{i/o}$, where for each $s \in (S \cup S \cdot A)$ and $e \in E$, $T_{i/o}(s \cdot e) = MO_{i/o}(s \cdot e)$

— The *hypothesis* can be computed out of the observation table as follows: $M : \mathcal{OT}_{i/o} \rightarrow \mathcal{DFA}$

- $\Sigma = \{row_{i/o}(s) | s \in S\}$,
- $\delta(row_{i/o}(s), a) = row_{i/o}(s \cdot a)$ $(s \in S, a \in A_I)$,
- $\chi(row_{i/o}(s), a) = T_{i/o}(s \cdot a)$ $(s \in S, a \in A_I)$,
- $s_0 = row_{i/o}(\lambda)$.

	$A\uparrow$	$B\uparrow$	$A\downarrow$	$B\downarrow$
λ	$init_A$	$init_B$	λ	λ
$A\uparrow$	λ	$init_B$	$clear_A$	λ
$B\uparrow$	$init_A$	λ	λ	$clear_B$
$A\downarrow$	$init_A$	$init_B$	λ	λ
$B\downarrow$	$init_A$	$init_B$	λ	λ
$A\uparrow\cdot A\uparrow$	λ	$init_B$	$clear_A$	λ
$\boxed{A\uparrow\cdot B\uparrow}$	λ	λ	$clear_A$	$clear_B$
$A\uparrow\cdot A\downarrow$	$init_A$	$init_B$	λ	λ
$A\uparrow\cdot B\downarrow$	λ	$init_B$	$clear_A$	λ
$\boxed{B\uparrow\cdot A\uparrow}$	λ	λ	$clear_A$	$clear_B$
$B\uparrow\cdot B\uparrow$	$init_A$	λ	λ	$clear_B$
$B\uparrow\cdot A\downarrow$	$init_A$	λ	λ	$clear_B$
$B\uparrow\cdot B\downarrow$	$init_A$	$init_B$	λ	λ

Again the observation table is not closed because of the two boxed rows. As they are equal, it is sufficient to add one of them to S.

	$A\uparrow$	$B\uparrow$	$A\downarrow$	$B\downarrow$
λ	$init_A$	$init_B$	λ	λ
$A\uparrow$	λ	$init_B$	$clear_A$	λ
$B\uparrow$	$init_A$	λ	λ	$clear_B$
$A\uparrow\cdot B\uparrow$	λ	λ	$clear_A$	$clear_B$
$A\downarrow$	$init_A$	$init_B$	λ	λ
$B\downarrow$	$init_A$	$init_B$	λ	λ
$A\uparrow\cdot A\uparrow$	λ	$init_B$	$clear_A$	λ
$A\uparrow\cdot A\downarrow$	$init_A$	$init_B$	λ	λ
$A\uparrow\cdot B\downarrow$	λ	$init_B$	$clear_A$	λ
$B\uparrow\cdot A\uparrow$	λ	λ	$clear_A$	$clear_B$
$B\uparrow\cdot B\uparrow$	$init_A$	λ	λ	$clear_B$
$B\uparrow\cdot A\downarrow$	$init_A$	λ	λ	$clear_B$
$B\uparrow\cdot B\downarrow$	$init_A$	$init_B$	λ	λ
$A\uparrow\cdot B\uparrow\cdot A\uparrow$	λ	λ	$clear_A$	$clear_B$
$A\uparrow\cdot B\uparrow\cdot B\uparrow$	λ	λ	$clear_A$	$clear_B$
$A\uparrow\cdot B\uparrow\cdot A\downarrow$	$init_A$	λ	λ	$clear_B$
$A\uparrow\cdot B\uparrow\cdot B\downarrow$	λ	$init_B$	$clear_A$	λ

Now the observation table is both closed and consistent. Furthermore it represents the final hypothesis. Note that we neither needed to extend the set E of distinguishing sequences, nor did we need to raise an equivalence query (except of the last one). The initial elements of E were sufficient to distinguish all the states. The final model is depicted in Figure 8.7.

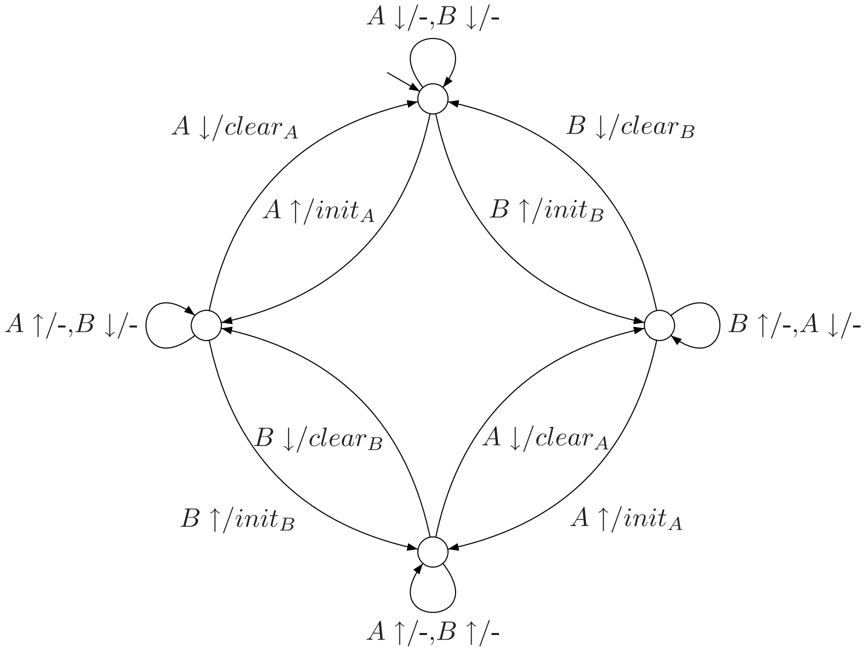


Figure 8.7: Final Input/Output Model for Scenario S_2

Results

During our experiments the assumption about the much better performance of $L^*_{i/o}$ in comparison to (optimized) L^* is confirmed. Note that we use the partial order and symmetry filters in both settings. Furthermore, because of the combination of inputs and the corresponding outputs at a single transition, instead of the introduction of intermediate states in case of the $\mathcal{DFA}$ representation, no distinction between the scenarios S_i and their counterparts S'_i is needed anymore. Additional output actions (*hookswitch*) do not result in additional states. Another observation during our experiments is that for all scenarios no call to the equivalence oracle is needed (except for the last one)[7]. Last but not least with our tailored algorithm $L^*_{i/o}$ we were even able to compute "bigger" scenarios, i.e. every involved device can hook off resp. hook on and every device can call each other (S_{Full}) and an additional, independent device is participating the setting ($S_{Full}+$). With the optimized L^* we were not able to compute these scenarios. The main reason for this phenomena is that in the $\mathcal{IODFA}$ representation we need 90 states, against 3236 states in the $\mathcal{DFA}$ representation. This, however, results in a much bigger observation table. Note that this is exactly the reason for the better performance of $L^*_{i/o}$ against L^* (cf. Table 8.3): as we need to distinguish fewer different states because of the tailored model representation, we need fewer different sequences in the set E. But each

[7]To arrive at comparable models that we have achieved during the run of L^*.

element of E requires "filling" a whole column of entries in the observation table, which usually leads to additional membership queries.

Table 8.3: Number of Membership Queries for Input/Output Learning

Scenario	States	$L^*_{i/o}$	States	L^*	Factor	Opt. L^*	Factor
S_1	2	10	5	108	10.8	15	1.5
S'_1	2	10	7	672	67.2	31	3.1
S_2	4	26	13	2,431	93.5	98	3.8
S'_2	4	26	29	15,425	593.3	145	5.6
S_3	8	42	33	19,426	462.5	207	4.9
S'_3	8	42	81	132,340	3,151.0	289	6.9
S_4	14	341	79	132,300	388.0	1,607	4.7
S_{Full}	90	7,343	-	-	-	-	-
$S_{Full}+$	180	16,421	-	-	-	-	-

Table 8.3 presents the practical results from our experiments. It can be seen that the reduction of membership queries with respect to the basic learning algorithm is enormous. But even in comparison to the (already) optimized setting we are able to reduce the number of membership queries by an average factor of about 4.4 (3.7 for the "normal" scenarios resp. 5.1 for the "primed" ones). Additionally we are able to compute even "bigger" scenarios.

8.5.2 Correctness and Termination of $L^*_{i/o}$

To show the correctness and termination of $L^*_{i/o}$ we can follow the corresponding proofs for L^* as presented in [Ang87], because the general structure of the algorithms is the same. We present the proofs, however, in a more transparent fashion, to allow reuse for proving future optimizations.

In the following it suffices to show that $L^*_{i/o}$ terminates, because whenever it terminates, it clearly produces the *correct result*. This is because of the assumption of the equivalence oracle.

In the rest of this section, $M(\mathcal{OT}_{i/o}) = (\Sigma, A_I, A_O, \delta, \chi, s_0)$ denotes the $\mathcal{IODFA}$ for $\mathcal{OT}_{i/o} = (S, E, T_{i/o})$, computed by the instructions of Algorithm 8.2. In a first step we have to show that every $M(\mathcal{OT}_{i/o})$ is well-defined and indeed *consistent* with $\mathcal{OT}_{i/o}$, where consistency denotes intuitively that every entry of the observation table can be observed in the model.

Lemma 8.13. *$M(\mathcal{OT}_{i/o})$ is well-defined.*

Proof. The set S is non-empty and prefix-closed, thus it must contain λ, so s_0 is defined. E is non-empty as initialized with A_I, i.e. $A_I \subseteq E$. Thus, if $s, s' \in S$ such that $row_{i/o}(s) = row_{i/o}(s')$, then for all $a \in A_I.T_{i/o}(s \cdot a) = T_{i/o}(s' \cdot a)$, so χ is well-defined. Because of $A_I \subseteq E$ the transition function δ is well-defined: suppose $s, s' \in S$ such that $row_{i/o}(s) = row_{i/o}(s')$. Then since $\mathcal{OT}_{i/o}$ is consistent, for each $a \in A_I$, $row_{i/o}(s \cdot a) = row_{i/o}(s' \cdot a)$ holds. Because $\mathcal{OT}_{i/o}$ is closed this common value is equal to $row_{i/o}(s'')$ for some $s'' \in S$. □

To prove that $M(\mathcal{OT}_{i/o})$ is consistent with $\mathcal{OT}_{i/o}$ we first need the following lemma which states that all access strings of the observation table are represented by valid states in M.

Lemma 8.14. Let $\mathcal{OT}_{i/o} = (S, E, T_{i/o})$ be a closed and consistent observation table. Then for $M(\mathcal{OT}_{i/o})$ the following holds, $\forall s \in (S \cup S \cdot A_I).\delta(s_0, s) = row_{i/o}(s)$.

Proof. We prove this lemma by induction on the length of s. For length 0, i.e. $s = \lambda$, it is obviously true, as $s_0 = row_{i/o}(\lambda)$.

Let as assume that for every $s \in (S \cup S \cdot A_I)$ of length at most $k \geq 0$, $\delta(s_0, s) = row_{i/o}(s)$ holds. Let $t \in (S \cup S \cdot A_I)$ with length $k+1$. Then we can find a decomposition of t such that $t = s \cdot a$ for some string s of length k and an element $a \in A_I$. This is because s must be in S as t is either in $S \cdot A$ (and therefore $s \in S$) or t is in S. But S is prefix-closed, thus $s \in S$. We can conclude

$$\begin{aligned}
\delta(s_0, t) &= \delta(s_0, s \cdot a) && (t = s \cdot a) \\
&= \delta(\delta(s_0, s), a) && \text{(Definition 8.4)} \\
&= \delta(row_{i/o}(s), a) && \text{(induction hypothesis)} \\
&= row_{i/o}(s \cdot a) && \text{(Algorithm 8.2, Definition of } M) \\
&= row_{i/o}(t) && (t = s \cdot a)
\end{aligned}$$

This completes the induction and the proof of Lemma 8.14. □

Note that from Lemma 8.14 we can also conclude that M is reduced, i.e. all states are reachable from within the start state.

Now we are ready to show that $M(\mathcal{OT}_{i/o})$ is consistent with $\mathcal{OT}_{i/o}$, or to be more precise with $T_{i/o}$.

Lemma 8.15. Let $\mathcal{OT}_{i/o} = (S, E, T_{i/o})$ be a closed and consistent observation table. Then $M(\mathcal{OT}_{i/o})$ is an $\mathcal{IODFA}$ *consistent* with the finite function $T_{i/o}$, i.e. $\forall s \in (S \cup S \cdot A_I), e \in E.\chi(\delta(s_0, s), e) = T_{i/o}(s \cdot e)$.

Proof. We prove this lemma by induction on the length of e. As E is initialized with A_I, we begin with length 1, i.e. $e \in A_I$ and $s \in (S \cup S \cdot A_I)$:

$$\begin{aligned}
\chi(\delta(s_0, s), e) &= \chi(row_{i/o}(s), e) && \text{(Lemma 8.14)} \\
&= T_{i/o}(s \cdot e) && \text{(Algorithm 8.2, Definition of } M\text{)}
\end{aligned}$$

Suppose that $\chi(\delta(s_0, s), e) = T_{i/o}(s \cdot e)$ holds for all elements $e \in E$ of length at most $k \geq 1$, and let $e' \in E$ be of length $k + 1$.

Now how can the set E evolve throughout the algorithm? The set E is initialized with the input alphabet, i.e. $A_I \subseteq E$, and is modified only when $\mathcal{OT}_{i/o}$ is not consistent. According to Definition 8.12.2 this means that two rows seems to be equivalent ($row_{i/o}(s_1) = row_{i/o}(s_2)$), but there exists a symbol a which separates them, i.e. leads to different rows ($row_{i/o}(s_1 \cdot a) \neq row_{i/o}(s_2 \cdot a)$). So there must exists an element e of E that can separate $row_{i/o}(s_1 \cdot a)$ and $row_{i/o}(s_2 \cdot a)$ and $a \cdot e$ will be added to the set E to separate $row_{i/o}(s_1)$ and $row_{i/o}(s_2)$. But this implies that for a member $e' \in E$ of length more than one we can find always a decomposition such that $e' = a \cdot e$ where e is of length k.
Let furthermore $s \in (S \cup S \cdot A_I)$. Then because $\mathcal{OT}_{i/o}$ is closed there exists a string $s' \in S$ such that $row_{i/o}(s) = row_{i/o}(s')$. Thus

$$\begin{aligned}
\chi(\delta(s_0, s), e') &= \chi(\delta(s_0, s), a \cdot e) && (e' = a \cdot e) \\
&= \chi(row_{i/o}(s), a \cdot e) && \text{(Lemma 8.14)} \\
&= \chi(row_{i/o}(s'), a \cdot e) && (row_{i/o}(s) = row_{i/o}(s')) \\
&= \chi(\delta(row_{i/o}(s'), a), e) && \text{(Definition 8.4)} \\
&= \chi(row_{i/o}(s' \cdot a), e) && \text{(Algorithm 8.2, Definition of } M\text{)} \\
&= \chi(\delta(s_0, s' \cdot a), e) && \text{(Lemma 8.14)} \\
&= T_{i/o}(s' \cdot a \cdot e) && \text{(induction hypothesis)} \\
&= T_{i/o}(s' \cdot e') && (e' = a \cdot e) \\
&= T_{i/o}(s \cdot e') && (row_{i/o}(s) = row_{i/o}(s'))
\end{aligned}$$

□

In the next step we have to prove that $M(\mathcal{OT}_{i/o})$ is the smallest $\mathcal{IODFA}$ consistent with $\mathcal{OT}_{i/o}$. Let us first define a relation ϕ that relates states of the hypothesis automaton $M(\mathcal{OT}_{i/o})$ to states from an arbitrary consistent automaton M'.

Lemma 8.16. Let $\mathcal{OT}_{i/o} = (S, E, T_{i/o})$ be a closed and consistent observation table and $M' = (\Sigma', A_I, A_O, \delta', \chi', s'_0)$ be any reduced $\mathcal{IODFA}$ consistent with $T_{i/o}$. Then the relation $\phi \subseteq \Sigma \times \Sigma'$, defined by $(row_{i/o}(s), \delta'(s'_0, s)) \in \phi$, is

1. left-total and

2. injective.

Proof. Let for each $s' \in \Sigma'$, $row'_{i/o}(s')$ be the finite function from E to A_O^*, such that $row'_{i/o}(e) = \sigma$ if and only if $\chi'(s', e) = \sigma$. From the consistency assumption, i.e. that M' is consistent with $T_{i/o}$, we can conclude that

$$\forall s \in (S \cup S \cdot A_I), e \in E.\chi'(\delta'(s'_0, s), e) = \sigma \text{ if and only if } T_{i/o}(s \cdot e) = \sigma$$

ad 1) To show that ϕ is left-total, we have to prove that for every state of $M(\mathcal{OT}_{i/o})$ at least one corresponding state in M' exists. From the consistency requirement we know that for every $s \in S$, $row_{i/o}(s)$ is equal to $row'_{i/o}(\delta'(s'_0, s))$, which implies particulary that for every row of $\mathcal{OT}_{i/o}$, and its corresponding state in $M(\mathcal{OT}_{i/o})$, a matching state in M' can be found.

ad 2) We can now prove that ϕ is injective, i.e. that every two states of $M(\mathcal{OT}_{i/o})$ that have a common state in their image are equal. Let $s_1, s_2 \in S$ and let us assume that they are mapped to the same state $s' \in \Sigma'$, i.e. $(row_{i/o}(s_1), s') \in \phi$ and $(row_{i/o}(s_2), s') \in \phi$, where $s' = \delta'(s'_0, s_1) = \delta'(s'_0, s_2)$ (Definition of ϕ). But this implies that $row_{i/o}(s_1) = row'_{i/o}(s')$ and $row_{i/o}(s_2) = row'_{i/o}(s')$ because of the consistency requirement. Thus if ϕ maps $row_{i/o}(s_1)$ and $row_{i/o}(s_2)$ both to state s', then the rows must be equal. $\square$

Corollary 8.17. $|M'| \leq |M(\mathcal{OT}_{i/o})| \Rightarrow \phi$ is functional.

Proof. A relation is functional if the following holds: $(a, b) \in \mathcal{R} \wedge (a, c) \in \mathcal{R} \Rightarrow b = c$. If M' has a less or equal number of states than $M(\mathcal{OT}_{i/o})$ it is easy to see that it must have exactly the number of states of $M(\mathcal{OT}_{i/o})$. The reason is that we know from Lemma 8.16 that for every state of $M(\mathcal{OT}_{i/o})$ a corresponding one in M' exists and that this mapping is injective. But as both M' and $M(\mathcal{OT}_{i/o})$ have a finite number of states, this implies that ϕ must be right-total and particulary be functional, which completes this proof. $\square$

Lemma 8.18. $|M'| \leq |M(\mathcal{OT}_{i/o})| \Rightarrow \phi$ is an isomorphism.

Proof. Because ϕ is one-to-one (Lemma 8.16) and onto (Corollary 8.17) it suffices to show that ϕ is a homomorphism. Therefore we must note that if $|M'| \leq |M(\mathcal{OT}_{i/o})|$, ϕ is indeed a well-defined function because we know that ϕ is left-total (Lemma 8.16) and functional (Corollary 8.17).

To show that ϕ is a homomorphism, we have to verify that it carries s_0 to s'_0 and that it preserves the transition function and output function.

To see that $\phi(s_0) = s'_0$ note that

$$\begin{aligned}\phi(s_0) &= \phi(row_{i/o}(\lambda)) \\ &= \delta'(s'_0, \lambda) \\ &= s'_0\end{aligned}$$

For each $s \in S$ and $a \in A_I$, let s_1 be an element of S such that $row_{i/o}(s \cdot a) = row_{i/o}(s_1)$. Then

$$\begin{aligned}\phi(\delta(row_{i/o}(s), a)) &= \phi(row_{i/o}(s \cdot a)) \\ &= \phi(row_{i/o}(s_1)) \\ &= \delta'(s'_0, s_1) \\ &= \delta'(s'_0, s \cdot a) && (*) \\ &= \delta'(\delta'(s'_0, s), a) \\ &= \delta'(\phi(row_{i/o}(s)), a)\end{aligned}$$

(*) Since $\delta'(s'_0, s_1)$ and $\delta'(s'_0, s \cdot a)$ have identical row values they must represent the same state of M'.

To finish the proof we must verify that the output function is preserved. But this is clear since

$$\begin{aligned}\chi(row_{i/o}(s), a) &= T_{i/o}(s \cdot a) \\ &= \chi'(\delta'(s'_0, s), a) \\ &= \chi'(row_{i/o}(s), a)\end{aligned}$$

□

Now we are ready to formulate the final result, i.e. that any minimal and reduced $\mathcal{IODFA}$ which is consistent with the observation table $\mathcal{OT}_{i/o}$, is unambiguously determined.

Theorem 8.19. *Let $\mathcal{OT}_{i/o} = (S, E, T_{i/o})$ be a closed and consistent observation table. Then $M(\mathcal{OT}_{i/o})$ is the smallest $\mathcal{IODFA}$ consistent with the finite function $T_{i/o}$.*

Proof. We know from Lemma 8.18 that any $\mathcal{IODFA}$ that is consistent with $T_{i/o}$ and has less or equal states than $M(\mathcal{OT}_{i/o})$ is isomorphic to $M(\mathcal{OT}_{i/o})$. Thus any other $\mathcal{IODFA}$ that is consistent with $\mathcal{OT}_{i/o}$ must have at least one more state. But this implies that $M(\mathcal{OT}_{i/o})$ is the uniquely smallest $\mathcal{IODFA}$ consistent with $\mathcal{OT}_{i/o}$. □

Termination of $L^*_{i/o}$

Now we are ready to show that $L^*_{i/o}$ does indeed terminate. As $L^*_{i/o}$ inherits the base algorithm from L^*, the argumentation about the correct termination is similar.

Theorem 8.20. *$L^*_{i/o}$ terminates.*

Proof. Let us assume that the correct (minimal) $\mathcal{IODFA}$ $M_U = (\Sigma, A_I, A_O, \delta, \chi, s_0)$ has exactly n states. We have to show that the number of distinct values of $row(s)$ for $s \in S$ increases strictly monotone up to a limit of n.

Let $\mathcal{OT}_{i/o} = (S, E, T_{i/o})$ denote a closed and consistent observation table. How can the set of state access strings S evolve throughout a run of the algorithm $L^*_{i/o}$? We have to distinguish three different situations in each step:

1. $\mathcal{OT}_{i/o}$ is not closed, or
2. $\mathcal{OT}_{i/o}$ is not consistent, or
3. a hypothesis is not correct.

We will show that in each of the above mentioned (internal) operations the number of distinct rows in S increases by at least one. Thus the total number of operations of either type over the whole run of $L^*_{i/o}$ must be at most $n-1$, since there is initially at least one value of $row(s)$ and there cannot be more than n.

ad 1) If $\mathcal{OT}_{i/o}$ is not closed, then there exists $row(s_1 \cdot a)$ such that for all $s \in S$, $row(s_1 \cdot a) \neq row(s)$. Thus $s_1 \cdot a$ will be added to S, which increases the number of distinct rows in S by at least one.

ad 2) If $\mathcal{OT}_{i/o}$ is not consistent, then there exists an input symbol $a \in A_I$, such that $row(s_1) = row(s_2)$ but $row(s_1 \cdot a) \neq row(s_2 \cdot a)$, i.e. $\exists e \in E. T_{i/o}(s_1 \cdot a \cdot e) \neq T_{i/o}(s_2 \cdot a \cdot e)$. So $a \cdot e$ will be added to E to distinguish between s_1 and s_2, which have been wrongly been classified as equivalent. But this implies that the number of distinct row in S increases, as all rows that have already been identified as being not equal remain unequal.

ad 3) If a hypothesis $M(\mathcal{OT}_{i/o})$ is found to be incorrect by a counterexample σ_c, then since the correct minimal $\mathcal{IODFA}$ M_U is consistent with $T_{i/o}$ and inequivalent to $M(\mathcal{OT}_{i/o})$, by Theorem 8.19, M_U must have at least one more state. I.e. $M(\mathcal{OT}_{i/o})$ has at most $n-1$ states. $L^*_{i/o}$ incorporates σ_c by adding the string σ_c

and all its prefixes to the set S. After bringing $\mathcal{OT}_{i/o}$ into a closed and consistent state, resulting in $\mathcal{OT}'_{i/o} = (S', E', T'_{i/o})$, $L^*_{i/o}$ makes a next hypothesis $M(\mathcal{OT}'_{i/o})$. On the one hand this hypothesis is still consistent with $T_{i/o}$, because it must classify each string $s \in (S \cup S \cdot A_I) \cdot E$ the same as before ($T'_{i/o}$ extends $T_{i/o}$). But on the other hand as σ_c is in S' and $A_I \subseteq E$ it classifies σ_c the same as M_U, i.e. $\forall a \in A_I . T'_{i/o}(\sigma_c \cdot a) = \chi(\sigma_c, a)$. This implies that $M(\mathcal{OT}'_{i/o})$ is inequivalent to $M(\mathcal{OT}_{i/o})$ as M_U is inequivalent to $M(\mathcal{OT}_{i/o})$. Therefore $M(\mathcal{OT}'_{i/o})$ must have at least one more state than $M(\mathcal{OT}_{i/o})$ (Theorem 8.19), i.e. S' has at least one "new" row in comparison to S.

To sum up this shows that $L^*_{i/o}$ can make a sequence of at most $n-1$ incorrect conjectures, since we have shown that the number of their states must be monotonically increasing. Since $L^*_{i/o}$ must, as long as it is running, eventually make another hypothesis, it must terminate by making a correct hypothesis.

$\square$

Chapter 9

Conclusions

This thesis concludes with a brief summary in the first section and an outlook for future work in the second section.

9.1 Summary

In this thesis we present a novel approach to the integrated testing of complex systems. Our approach covers the whole system-level test process, i.e. test case design, test case verification, test case execution, and test run analysis. Test cases can be designed from building-blocks, representing generic test actions resp. observations. Test engineers are guided during the design phase by online verification, where test cases are constantly checked against consistency rules. These rules can be formulated with the help of a pattern-system, which allows test engineers to specify them easily. In case of an inconsistency, diagnostic information is presented to the test engineer to guide the test case design. Furthermore, test cases can be executed within the test environment. For this purpose we provide a precise execution semantic of test cases. During test execution a detailed report will be prepared in the form of a test protocol, used for test run analysis. To sum up, this approach enables us to shift the test design issues from experts of the system and the used test tools to experts in the systems's logic only.

Our integrated test approach is supported by the *Integrated Test Environment* (*ITE*), a modular and open test framework. The *ITE* supports a uniform handling of the involved test tools and provides a well-defined and tool-supported integration process on the basis of an interface description of the test tool. The interface description can be used to automatically generate the test building-blocks, used for test case design. Thus test tool vendors only have to "export" the required functionality via CORBA. In this way we are in principle able to coordinate almost every test tool. In addition the CORBA-based communication layer supports the required distributed test execution.

We have illustrated the use of the *ITE* in several industrial case studies. In particular the field experience in testing *Computer Telephony Integrated* solutions (*CTI*) gained in a project with Siemens AG [Sie] has shown that the usage of the *ITE* has brought a new dimension of efficiency (with measured speedup factors of above 30 for the test execution phase). A global cost-benefit calculation thus shows that the additional investment for the *ITE* pays for itself in a short period of time, if extensively adopted. The *ITE* in fact, dramatically reduces the recurring cost factors without impairing the remaining positions involving the basic effort that still has to be taken during the whole test lifecycle (test planning, manual configuration of the test settings, ...) and the necessary up-front investments (e.g. license fees for test tools, hardware, ...).
In the context of *Web-based* applications the *ITE* has also been successfully applied. Because of the restricted and standardized set of available GUI controls for such applications, a high reuse factor can be achieved. Once a web-based application is integrated into the *ITE* almost every other web-based application can be tested with the same set of test building-blocks.

Finally, we have presented an approach for a posteriori model generation for complex systems. The resulting models can never be exact, i.e. reflect the complete and correct behaviour of the considered system. Nevertheless they can be useful in practice, to represent the cumulative knowledge of the system in a consistent description. For the model generation procedure we have optimized a standard learning method according to domain-specific structural properties. Two different optimized versions of the learning algorithm are presented:

— L^* *with Filters*: Modifying the learning algorithm by filtering out unnecessary queries. We incorporated application-specific characteristics of the considered scenario, i.e. prefix-closeness, input-determinism, as well as partial order and symmetries between events. These filters provides a flexible approach which permits fast adaptations to different application domains.

— $L^*_{i/o}$: Tailored to deal with input/output deterministic systems this algorithm incorporates already some of the filters (prefix-closeness and input-determinism), while others are still implemented via filters (partial order and symmetries between events). With this optimized version we are able to improve the learning efficiency in comparison to L^* with filters once more. The main reason for this effect is that we use a specialized representation for these systems (*input/output deterministic finite automata*) which reduces the number of states needed for the representation.

We have demonstrated that with the right modifications to the learning process and the right environment for observing systems, non-trivial system models may in fact be learned in practice.

9.2 Future Work

This section indicates some directions of future research.

Concurrent Tests

In our experience so far no notation of concurrency is needed for the definition of system-level tests. It would, however, be an interesting issue because the *ITE* provides a distributed test execution environment and it is therefore in principle able to execute concurrent tests. This can improve the execution of special sorts of tests, e.g. where independency between parts of a test exist. Two special types of test blocks can be responsible for specifying concurrency: one for splitting the control flow and another one for synchronization. For test execution some properties of the test cases have to be ensured:

- The concurrently executed parts of a test case must be independent, i.e. do not affect each other. This can be ensured, e.g. when the actions operates on different address ranges (cf. Section 8.4).
- To ensure "true" concurrency the concurrent executed parts of a test case must be executed by different test tools.

In Figure 9.1 this approach is illustrated for a simple example in the context of telephony systems (cf. Figure 8.5). The original, sequential test case can be seen on the left side: a call between device A and device B will be established and an additional device C performs an *offHook/onHook* action. All actions are executed by the test tool $Tool_1$. The test case comprises of two independent sequences, as the call between device A and B is independent of device C. This means that it is possible to transform the sequential test case into a concurrent one (see Figure 9.1 (right)). The special test block *conc* states that the subsequent test blocks are executed concurrently until the control flow synchronizes again through the *sync* test block. To provide "true" concurrency the two parallel parts of the test case must be executed by different test tools, i.e. the right path will be executed by the test tool $Tool_2$.

It is in principle possible to compute the independent parts of a test case automatically, as we have illustrated in Chapter 8. The distribution among the involved

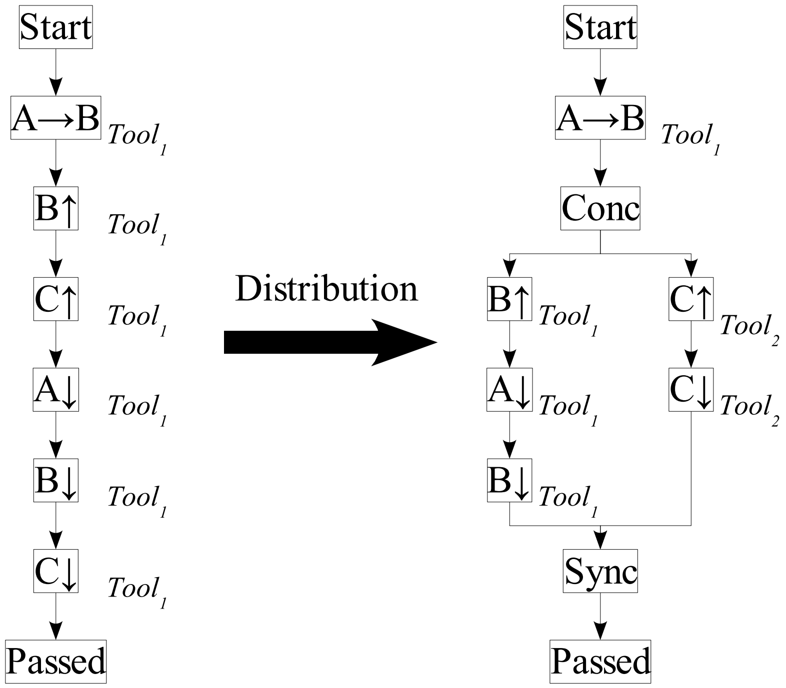


Figure 9.1: Distribution of Concurrent Tests

test tools, however, is a non-trivial task (cf. e.g. [JJKV98, TVJ00]). A practical approach may be to define a mapping of address ranges to the different test tools.

Learning Partial Order Reduced Models

Another powerful optimization that remains is to learn partial order reduced models instead of learning the complete model. The advantage will be that a reduced model is usually much smaller in the number of states than the complete model, which will again improve the performance of the learning process. Furthermore, one is able to reverse the partial order reduction to complete the model again. Thus the key idea is: learn the smaller model with the "expensive" learning process and expand it on demand to the complete model. Note that for many applications it is sufficient to consider the partial order reduced model only, e.g. for checking deadlocks, safety properties or LTL formulae [God94]

In Figure 9.2 the general idea is sketched. Imagine that actions a and b are independent of each other. In a partial order reduced model we only want to incorporate one representative trace, e.g. $a \cdot b$. All other possible orderings should be suppressed, e.g. in the example depicted in Figure 9.2 the sequence $b \cdot a$ will not be incorporated, i.e. we "cut" the b-transition. The example of Figure 9.2 indicates that in general partial

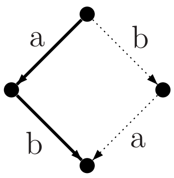


Figure 9.2: Normalized Trace

order reduction is ambiguous. This is because it is sufficient to consider one element of the equivalence class only, but, as all are equal, it does not matter which one we choose. For our learning algorithm L^*, however, it is necessary that the reduction is unambiguous. This is because the algorithm relies on reproducible results. So what we need is a *normalization function* η that normalizes every trace and for every sequence of its equivalence class always returns a unique representative of this class. It will depend on a concrete implementation which unique representative will be chosen. Note, however, that this problem is non-trivial, as in general the language induced by the set of normalized traces for a system need not be regular [Maz87]. For a simple example just consider the language $\mathcal{L} = (ab)^*$ and a normalization η, where all a's occur before the b's. Then the language induced by the normalization η is given by $a^n b^n$ which is not regular.

For an implementation of this algorithm (L^*_η) along the ideas of L^*, we could "cut" the model at those transitions where it "leaves" the partial order reduced model. For this purpose all rows s where the consistency requirement $s = \eta(s)$ does not hold, could be marked with the special output symbol $\perp$. In consequence the "core" algorithm L^* would merge all states that do not belong to the partial order reduced submodel, into one single state. For the computation of a hypothesis out of the observation table we simply have to disregard this special state.

Basically the algorithm L^*_η and $L^*_{i/o}$ can operate on the same observation table. However, for L^*_η we have to enrich the output alphabet to capture disallowed paths. Remarkable differences to $L^*_{i/o}$ would then be found in the *update* procedure of the observation table, where for all rows that are not consistent with η, all entries in this row are set to a special symbol ($\perp$). These rows can then be disregarded during the computation of the hypothesis, which yields to a partial $\mathcal{IODFA}$ (the transition function δ and the output function χ are partially defined only). Note that if the observation table is filled according these instructions, it is possible that two rows can be distinguished by some element e out of E, but for some other row s (where $s = \eta(s)$) $s \cdot e \neq \eta(s \cdot e)$ holds. To arrive at a complete observation table, i.e. without "holes", we have to fill these entries as usual. Furthermore, we would need a special equivalence oracle that conforms to the normalization function η and induces a partial order reduced model, the completion of which is indeed the full model.

First experiments with an implementation for the algorithm L^*_η emphasizes the great impact. Already in our first measurements ($S_{Full}+$) we are able to improve the performance by approximately 30% with respect to $L^*_{i/o}$ (with partial order and symmetry filters). We expect, however, that this factor increases for larger systems.

LearnAutomaton

An improved version of L^* is called LearnAutomaton [KV94]. It uses a binary classification tree rather than an observation table as the central datastructure. The leaves of the tree represent the state rows and the internal nodes the distinguishing sequences (E). Then any two states, i.e. leaves in the classification tree, are distinguished by the string labelling of their least common ancestor in the classification tree. Transitions have to be found via a look-up (*sift*) in the classification tree for a state string extended by all alphabet symbols by membership queries.

On the one hand the classification tree reduces the number of membership queries, because we do not have to apply unnecessary elements of E to all state strings. On the other hand this approach requires that the equivalence question is raised exactly n times, if n denotes the number of states of the target automaton. This must be seen as a drawback in our case where the membership queries are reliable but the equivalence queries are not (approximate equivalence oracle). However, one can think of maintaining a partial or reduced observation table by incorporating the ideas of LearnAutomaton: It is sufficient to have one witness of inequality for each pair of state strings. This will lead to an observation table, where only the cell entries are computed that are needed to distinguish between states. The others can be computed on demand if they are needed later during the learning procedure. In any case it will be necessary to adapt the notations of *closeness* and *consistency* to deal with the incomplete rows.

Part V

Appendices

Appendix A

First-order property pattern mappings for *ESLTL*

In the following we present a mapping of the property specification patterns for our first order extension of the semantic linear-time logic. The concrete second dimension, i.e. the quantification, is denoted through $\mathcal{Q} = \{\exists, \forall\}$.

Absence

P is `false`:

Globally	$\mathcal{Q}x(\mathbf{G}(\neg P(x_p)))$
Before R	$\mathcal{Q}x(\mathbf{F}(R(x_r)) \Rightarrow (\neg P(x_p)\ \mathbf{U}\ R(x_r)))$
After Q	$\mathcal{Q}x(\mathbf{G}(Q(x_q) \Rightarrow \mathbf{G}(\neg P(x_p))))$
Between Q and R	$\mathcal{Q}x(\mathbf{G}((Q(x_q) \wedge \neg R(x_r) \wedge \mathbf{F}(R(x_r))) \Rightarrow (\neg P(x_p)\ \mathbf{U}\ R(x_r))))$
After Q until R	$\mathcal{Q}x(\mathbf{G}(Q(x_q) \wedge \neg R(x_r) \Rightarrow (\neg P(x_p)\ \mathbf{WU}\ R(x_r))))$

Universality

P is `true`:

Globally	$\mathcal{Q}x(\mathbf{G}(P(x_p)))$
Before R	$\mathcal{Q}x(\mathbf{F}(R(x_r)) \Rightarrow (P(x_p)\ \mathbf{U}\ R(x_r)))$
After Q	$\mathcal{Q}x(\mathbf{G}(Q(x_q) \Rightarrow \mathbf{G}P(x_p)))$
Between Q and R	$\mathcal{Q}x(((Q(x_q) \wedge \neg R(x_r) \wedge \mathbf{F}(R(x_r))) \Rightarrow (P(x_p)\ \mathbf{U}\ R(x_r))))$
After Q until R	$\mathcal{Q}x(\mathbf{G}(Q(x_q) \wedge \neg R(x_r) \Rightarrow (P(x_p)\ \mathbf{WU}\ R(x_r))))$

Existence

P is `true`:

Globally	$\mathcal{Q}x(\mathbf{F}(P(x_p)))$
Before R	$\mathcal{Q}x(\neg R(x_r)\ \mathbf{WU}\ (P(x_p) \wedge \neg R(x_r)))$
After Q	$\mathcal{Q}x(\mathbf{G}(\neg Q(x_q) \vee \mathbf{F}(Q(x_q) \wedge \mathbf{F}(P(x_p)))))$
Between Q and R	$\mathcal{Q}x(\mathbf{G}(Q(x_q) \wedge \neg R(x_r) \Rightarrow (\neg R(x_r)\ \mathbf{WU}\ (P(x_p) \wedge \neg R(x_r)))))$
After Q until R	$\mathcal{Q}x(\mathbf{G}(Q(x_q) \wedge \neg R(x_r) \Rightarrow (\neg R(x_r)\ \mathbf{U}\ (P(x_p) \wedge \neg R(x_r)))))$

Precedence

S precedes P:

Globally	$\mathcal{Q}x(\neg P(x_p)\ \mathbf{WU}\ S(x_s))$
Before R	$\mathcal{Q}x(\mathbf{F}(R(x_r)) \Rightarrow (\neg P(x_p)\ \mathbf{U}\ (S(x_s) \vee R(x_r))))$
After Q	$\mathcal{Q}x(\mathbf{G}(\neg Q(x_q)) \vee \mathbf{F}(Q(x_q) \wedge (\neg P(x_p)\ \mathbf{WU}\ S(x_s))))$
Between Q and R	$\mathcal{Q}x(\mathbf{G}((Q(x_q) \wedge \neg R(x_r) \wedge \mathbf{F}(R(x_r))) \Rightarrow (\neg P(x_p)\ \mathbf{U}\ (S(x_s) \vee R(x_r)))))$
After Q until R	$\mathcal{Q}x(\mathbf{G}(Q(x_q) \wedge \neg R(x_r) \Rightarrow (\neg P(x_p)\ \mathbf{WU}\ (S(x_s) \vee R(x_r)))))$

Response

S responds to P:

Globally	$\mathcal{Q}x(\mathbf{G}(P(x_p) \Rightarrow \mathbf{F}(S(x_s))))$
Before R	$\mathcal{Q}x(\mathbf{F}(R(x_r)) \Rightarrow (P(x_p) \Rightarrow (\neg R(x_r)\ \mathbf{U}\ (S(x_s) \wedge \neg R(x_r))))\ \mathbf{U}\ R(x_r))$
After Q	$\mathcal{Q}x(\mathbf{G}(Q(x_q) \Rightarrow \mathbf{G}(P(x_p) \Rightarrow \mathbf{F}(S(x_s)))))$
Between Q and R	$\mathcal{Q}x(\mathbf{G}((Q(x_q) \wedge \neg R(x_r) \wedge \mathbf{F}(R(x_r))) \Rightarrow (P(x_p) \Rightarrow (\neg R(x_r)\ \mathbf{U}\ (S(x_s) \wedge \neg R(x_r))))\ \mathbf{U}\ R(x_r)))$
After Q until R	$\mathcal{Q}x(\mathbf{G}(Q(x_q) \wedge \neg R(x_r) \Rightarrow ((P(x_p) \Rightarrow (\neg R(x_r)\ \mathbf{U}\ (S(x_s) \wedge \neg R(x_r))))\ \mathbf{WU}\ R(x_r)))$

Bounded Existence

Instance of bounded existence pattern, where the bound is at most two designated states. Transitions to P-states occur at most 2 times:

Globally	$\mathcal{Q}x((\neg P(x_p)\ \mathbf{WU}\ (P(x_p)\ \mathbf{WU}\ (\neg P(x_p)\ \mathbf{WU}\ (P(x_p)\ \mathbf{WU}\ \mathbf{G}(\neg P(x_p)))))))$
Before R	$\mathcal{Q}x(\mathbf{F}(R(x_r)) \Rightarrow ((\neg P(x_p) \wedge \neg R(x_r))\ \mathbf{U}\ (R(x_r) \vee ((P(x_p) \wedge \neg R(x_r))\ \mathbf{U}\ (R(x_r) \vee ((\neg P(x_p) \wedge \neg R(x_r))\ \mathbf{U}\ (R(x_r) \vee ((P(x_p) \wedge \neg R(x_r))\ \mathbf{U}\ (R(x_r) \vee (\neg P(x_p)\ \mathbf{U}\ R(x_r))))))))))$
After Q	$\mathcal{Q}x(\mathbf{F}(Q(x_q)) \Rightarrow (\neg Q(x_q)\ \mathbf{U}\ (Q(x_q) \wedge (\neg P(x_p)\ \mathbf{WU}\ (P(x_p)\ \mathbf{WU}\ (\neg P(x_p)\ \mathbf{WU}\ (P(x_p)\ \mathbf{WU}\ \mathbf{G}(\neg P(x_p))))))))))$
Between Q and R	$\mathcal{Q}x(\mathbf{G}((Q(x_q) \wedge \mathbf{F}(R(x_r))) \Rightarrow ((\neg P(x_p) \wedge \neg R(x_r))\ \mathbf{U}\ (R(x_r) \vee ((P(x_p) \wedge \neg R(x_r))\ \mathbf{U}\ (R(x_r) \vee ((\neg P(x_p) \wedge \neg R(x_r))\ \mathbf{U}\ (R(x_r) \vee ((P(x_p) \wedge \neg R(x_r))\ \mathbf{U}\ (R(x_r) \vee (\neg P(x_p)\ \mathbf{U}\ R(x_r)))))))))))$
After Q until R	$\mathcal{Q}x(\mathbf{G}(Q(x_q) \Rightarrow ((\neg P(x_p) \wedge \neg R(x_r))\ \mathbf{U}\ (R(x_r) \vee ((P(x_p) \wedge \neg R(x_r))\ \mathbf{U}\ (R(x_r) \vee ((\neg P(x_p) \wedge \neg R(x_r))\ \mathbf{U}\ (R(x_r) \vee ((P(x_p) \wedge \neg R(x_r))\ \mathbf{U}\ (R(x_r) \vee (\neg P(x_p)\ \mathbf{WU}\ R(x_r)) \vee \mathbf{G}(P(x_p))))))))))))$

Precedence Chain

S, T precedes P (2 cause-1 effect precedence chain):

Globally	$\mathcal{Q}x(\mathbf{F}(P(x_p)) \Rightarrow (\neg P(x_p)\ \mathbf{U}\ (S(x_s) \wedge \neg P(x_p) \wedge \mathbf{X}(\neg P(x_p)\ \mathbf{U}\ T(x_t)))))$
Before R	$\mathcal{Q}x(\mathbf{F}(R(x_r)) \Rightarrow (\neg P(x_p)\ \mathbf{U}\ (R(x_r) \vee (S(x_s) \wedge \neg P(x_p) \wedge X(\neg P(x_p)\ \mathbf{U}\ T(x_t))))))$
After Q	$\mathcal{Q}x((\mathbf{G}(\neg Q(x_q))) \vee (\neg Q(x_q)\ \mathbf{U}\ (Q(x_q) \wedge \mathbf{F}(P(x_p)) \Rightarrow (\neg P(x_p)\ \mathbf{U}\ (S(x_s) \wedge \neg P(x_p) \wedge \mathbf{X}(\neg P(x_p)\ \mathbf{U}\ T(x_t)))))))$
Between Q and R	$\mathcal{Q}x(\mathbf{G}((Q(x_q) \wedge \mathbf{F}(R(x_r))) \Rightarrow (\neg P(x_p)\ \mathbf{U}\ (R(x_r) \vee (S(x_s) \wedge \neg P(x_p) \wedge \mathbf{X}(\neg P(x_p)\ \mathbf{U}\ T(x_t)))))))$
After Q until R	$\mathcal{Q}x(\mathbf{G}(Q(x_q) \Rightarrow (\mathbf{F}(P(x_p)) \Rightarrow (\neg P(x_p)\ \mathbf{U}\ (R(x_r) \vee (S(x_s) \wedge \neg P(x_p) \wedge \mathbf{X}(\neg P(x_p)\ \mathbf{U}\ T(x_t)))))))$

P precedes (S, T) (1 cause-2 effect precedence chain):

Globally	$\mathcal{Q}x((\mathbf{F}(S(x_s) \wedge \mathbf{X}(\mathbf{F}(T(x_t)))) \Rightarrow (\neg S(x_s)\ \mathbf{U}\ P(x_p))))$
Before R	$\mathcal{Q}x(\mathbf{F}(R(x_r)) \Rightarrow ((\neg(S(x_s) \wedge \neg R(x_r) \wedge \mathbf{X}(\neg R(x_r)\ \mathbf{U}\ (T(x_t) \wedge \neg R(x_r))))) \ \mathbf{U}\ (R(x_r) \vee P(x_p))))$
After Q	$\mathcal{Q}x((\mathbf{G}(\neg Q(x_q))) \vee (\neg Q(x_q)\ \mathbf{U}\ (Q(x_q) \wedge ((\mathbf{F}(S(x_s) \wedge \mathbf{X}(\mathbf{F}(T(x_t))))) \Rightarrow (\neg S(x_s)\ \mathbf{U}\ P(x_p))))))$
Between Q and R	$\mathcal{Q}x(\mathbf{G}((Q(x_q)\wedge\mathbf{F}(R(x_r))) \Rightarrow ((\neg(S(x_s)\wedge\neg R(x_r)\wedge\mathbf{X}(\neg R(x_r)\ \mathbf{U}\ (T(x_t)\wedge\neg R(x_r))))) \ \mathbf{U}\ (R(x_r)\vee P(x_p)))))$
After Q until R	$\mathcal{Q}x(\mathbf{G}(Q(x_q) \Rightarrow (\neg(S(x_s) \wedge \neg R(x_r) \wedge \mathbf{X}(\neg R(x_r)\ \mathbf{U}\ (T(x_t) \wedge \neg R(x_r)))) \ \mathbf{U}\ (R(x_r) \vee P(x_p)) \vee \mathbf{G}(\neg(S(x_s) \wedge \mathbf{X}(\mathbf{F}(T(x_t))))))))$

Response Chain

P responds to S, T (2 stimulus-1 response chain):

Globally	$\mathcal{Q}x(\mathbf{G}(S(x_s) \wedge \mathbf{X}(\mathbf{F}(T(x_t))) \Rightarrow \mathbf{X}(\mathbf{F}(T(x_t) \wedge \mathbf{F}(P(x_p))))))$
Before R	$\mathcal{Q}x(\mathbf{F}(R(x_r)) \Rightarrow (S(x_s) \wedge \mathbf{X}(\neg R(x_r)\ \mathbf{U}\ T(x_t)) \Rightarrow \mathbf{X}(\neg R(x_r)\ \mathbf{U}\ (T(x_t) \wedge \mathbf{F}(P(x_p))))) \ \mathbf{U}\ R(x_r))$
After Q	$\mathcal{Q}x(\mathbf{G}(Q(x_q) \Rightarrow \mathbf{G}(S(x_s) \wedge \mathbf{X}(\mathbf{F}(T(x_t))) \Rightarrow \mathbf{X}(\neg T(x_t)\ \mathbf{U}\ (T(x_t) \wedge \mathbf{F}(P(x_p)))))))$
Between Q and R	$\mathcal{Q}x(\mathbf{G}((Q(x_q) \wedge \mathbf{F}(R(x_r))) \Rightarrow (S(x_s) \wedge \mathbf{X}(\neg R(x_r)\ \mathbf{U}\ T(x_t)) \Rightarrow \mathbf{X}(\neg R(x_r)\ \mathbf{U}\ (T(x_t) \wedge \mathbf{F}(P(x_p))))) \ \mathbf{U}\ R(x_r)))$
After Q until R	$\mathcal{Q}x(\mathbf{G}(Q(x_q) \Rightarrow (S(x_s) \wedge \mathbf{X}(\neg R(x_r)\ \mathbf{U}\ T(x_t)) \Rightarrow \mathbf{X}(\neg R(x_r)\ \mathbf{U}\ (T(x_t) \wedge \mathbf{F}(P(x_p))))) \ \mathbf{U}\ (R(x_r) \vee \mathbf{G}(S(x_s) \wedge \mathbf{X}(\neg R(x_r)\ \mathbf{U}\ T(x_t)) \Rightarrow \mathbf{X}(\neg R(x_r)\ \mathbf{U}\ (T(x_t) \wedge \mathbf{F}(P(x_p))))))))$

S, T responds to P (1 stimulus-2 response chain):

Globally	$\mathcal{Q}x(\mathbf{G}(P(x_p) \Rightarrow \mathbf{F}(S(x_s) \wedge \mathbf{X}(\mathbf{F}(T(x_t))))))$
Before R	$\mathcal{Q}x(\mathbf{F}(R(x_r)) \Rightarrow (P(x_p) \Rightarrow (\neg R(x_r)\ \mathbf{U}\ (S(x_s) \wedge \neg R(x_r) \wedge \mathbf{X}(\neg R(x_r)\ \mathbf{U}\ T(x_t))))) \ \mathbf{U}\ R(x_r))$
After Q	$\mathcal{Q}x(\mathbf{G}(Q(x_q) \Rightarrow \mathbf{G}(P(x_p) \Rightarrow (S(x_s) \wedge \mathbf{X}(\mathbf{F}(T(x_t)))))))$
Between Q and R	$\mathcal{Q}x(\mathbf{G}((Q(x_q) \wedge \mathbf{F}(R(x_r))) \Rightarrow (P(x_p) \Rightarrow (\neg R(x_r)\ \mathbf{U}\ (S(x_s) \wedge \neg R(x_r) \wedge \mathbf{X}(\neg R(x_r)\ \mathbf{U}\ T(x_t))))) \ \mathbf{U}\ R(x_r)))$
After Q until R	$\mathcal{Q}x(\mathbf{G}(Q(x_q) \Rightarrow (P(x_p) \Rightarrow (\neg R(x_r)\ \mathbf{U}\ (S(x_s) \wedge \neg R(x_r) \wedge \mathbf{X}(\neg R(x_r)\ \mathbf{U}\ T(x_t))))) \ \mathbf{U}\ (R(x_r) \vee \mathbf{G}(P(x_p) \Rightarrow (S(x_s) \wedge \mathbf{X}(\mathbf{F}(T(x_t))))))))$

Appendix B

Document Type Descriptions for XML

B.1 DTD's for the *ITE* Constraint Editor

B.1.1 DTD for the Collection of Logic Patterns

```
<!-- The collection of logic pattern contains at least   -->
<!-- one pattern                                          -->
<!ELEMENT patternSystem (pattern+)>

<!-- raw-tags and placeholders toggles                    -->
<!ELEMENT pattern (raw | (p | q | r | s | t | u | v | w))*>
<!ATTLIST pattern name CDATA #REQUIRED
                  scope CDATA #REQUIRED>

<!ELEMENT raw (#PCDATA)>

<!-- The placeholders                                     -->
<!ELEMENT p EMPTY>
<!ELEMENT q EMPTY>
<!ELEMENT r EMPTY>
<!ELEMENT s EMPTY>
<!ELEMENT t EMPTY>
<!ELEMENT u EMPTY>
<!ELEMENT v EMPTY>
<!ELEMENT w EMPTY>
```

B.1.2 DTD for Composite Patterns

```
<!-- Composite Pattern consists of a description,        -->
<!-- the input description and the rules how to map it   -->
<!-- to the logic patterns                               -->
<!ELEMENT CompositePattern (patternDescription,
                       input,
                       generateLogicPattern?)>
<!ATTLIST CompositePattern name CDATA #REQUIRED
                      class CDATA #REQUIRED>

<!ELEMENT patternDescription (#PCDATA)>

<!-- Die input description consists of several steps     -->
<!ELEMENT input (step+)>

<!-- An input step consists out of its description       -->
<!-- and the corresponding input dialog.                 -->
<!ELEMENT step (stepDescription,
                dialog)>
<!ATTLIST step no CDATA #REQUIRED>

<!ELEMENT stepDescription (#PCDATA)>
<!ELEMENT dialog (#PCDATA)>

<!-- Two types of links are needed                       -->
<!ELEMENT simpleLink EMPTY>
<!ATTLIST simpleLink step CDATA #REQUIRED>

<!ELEMENT advancedLink (#PCDATA)>
<!ATTLIST advancedLink step CDATA #REQUIRED>

<!-- The generation of constraints                       -->
<!ELEMENT generateLogicPattern (logicPattern+)>

<!-- For a constraint the "name" and the corresponding   -->
<!-- pattern are needed. It consists of a description    -->
<!-- and a declaration how to generate the placeholders  -->
<!-- for the logic pattern                               -->
<!ELEMENT logicPattern (logicPatternDescription, sibs)>
<!ATTLIST logicPattern type (single|multiple) #REQUIRED
```

```
                       name CDATA #REQUIRED
                       pattern CDATA #REQUIRED
                       scope CDATA #REQUIRED>

<!ELEMENT logicPatternDescription (#PCDATA)>

<!-- The placeholders for the test blocks in the pattern -->
<!-- system                                              -->
<!ELEMENT sibs (p?,q?,r?,s?,t?,u?,v?,w?)>

<!ENTITY % sib "(name, (arg* | args))">

<!ELEMENT p %sib; >
<!ELEMENT q %sib; >
<!ELEMENT r %sib; >
<!ELEMENT s %sib; >
<!ELEMENT t %sib; >
<!ELEMENT u %sib; >
<!ELEMENT v %sib; >
<!ELEMENT w %sib; >

<!-- The name of a test block can be given directly      -->
<!-- or through a link                                   -->
<!ELEMENT name (#PCDATA | advancedLink)>

<!-- <arg> generates the test block parameter "name"     -->
<!-- The value of the parameter can be given directly    -->
<!-- or can be specified through a link                  -->
<!ELEMENT arg (#PCDATA | advancedLink)*>
<!ATTLIST arg name CDATA #REQUIRED
              required (yes|no) "yes">

<!-- <args> takes all <name>/<content>-pairs out of      -->
<!-- a certain step and generates parameters for a test block -->
<!ELEMENT args (simpleLink)>
```

B.1.3 DTD for concrete Constraints

```
<!-- A concrete Constraint consists of a description     -->
<!-- and serveral steps.                                 -->
```

```
<!ELEMENT constraint (description, step*)>

<!-- The Composite Pattern and the name of the constraint-->
<!-- will be given as arguments                          -->
<!ATTLIST constraint pattern CDATA #REQUIRED
                     name CDATA #REQUIRED>

<!ELEMENT description  (#PCDATA)>

<!ELEMENT step (nameSib,
                parameter*)>

<!ATTLIST step no CDATA #REQUIRED>

<!ELEMENT nameSib (#PCDATA)>

<!-- Parameters of test blocks                           -->
<!ELEMENT parameter (name,content)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT content (#PCDATA | forAll | exists)*>
<!ELEMENT forAll EMPTY>
<!ATTLIST forAll placeholder CDATA #REQUIRED>
<!ELEMENT exists EMPTY>
<!ATTLIST exists placeholder CDATA #REQUIRED>
```

B.2 DTD for *ITE* Test Reports

```
<!ELEMENT Testreport (Testinformation, Testlog?, Testresult)>
<!ELEMENT Testinformation (Testexecution, Testconfiguration,
                           Testscenario)>
<!ELEMENT Testexecution (DateandTimeofTest, SUTName,
                         Testcoordrevision?, Testlab?, Testengineer?)>
<!ELEMENT DateandTimeofTest (Date, Time)>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT Time (#PCDATA)>
<!ELEMENT SUTName (#PCDATA)>
<!ELEMENT Testcoordrevision (#PCDATA)>
<!ELEMENT Testlab (#PCDATA)>
<!ELEMENT Testengineer (#PCDATA)>
```

```
<!ELEMENT Testconfiguration (Testenvironment+, Systemenvironment+)>
<!ELEMENT Systemenvironment (Systemcomponent | Errormessage)>
<!ELEMENT Systemcomponent (Componentname, Configuration*)>
<!ELEMENT Componentname (#PCDATA)>
<!ELEMENT Configuration (#PCDATA)>
<!ATTLIST Configuration
    Configparameter CDATA #REQUIRED
    IsRevisioned CDATA #REQUIRED
    Revision CDATA #IMPLIED
>
<!ELEMENT Testenvironment (Testtool | Errormessage)>
<!ELEMENT Testtool (Tooldescription, Comment?, Configuration?)>
<!ELEMENT Tooldescription (#PCDATA)>
<!ATTLIST Tooldescription
    ToolId CDATA #REQUIRED
    Toolversion CDATA #IMPLIED
    ToolIPAddress CDATA #IMPLIED
>
<!ELEMENT Comment (#PCDATA)>
<!ELEMENT Testscenario (Scenarioname, Scenariodescription?)>
<!ELEMENT Scenarioname (#PCDATA)>
<!ATTLIST Scenarioname
    isRevisioned CDATA #IMPLIED
    Revision CDATA #IMPLIED
>
<!ELEMENT Scenariodescription (#PCDATA)>
<!ELEMENT Testlog (Logentry+)>
<!ELEMENT Logentry (Executiontime, Execelement, Execinfo?)>
<!ELEMENT Executiontime (#PCDATA)>
<!ELEMENT Execelement (#PCDATA)>
<!ATTLIST Execelement
    Exectype CDATA #REQUIRED
    Revision CDATA #IMPLIED
>
<!ELEMENT Execinfo (SIBName, SIBActivator, Actualparameter*,
                    SIBDatablock?, Branchexecution?)>
<!ELEMENT SIBName (#PCDATA)>
<!ATTLIST SIBName
    SIBId CDATA #IMPLIED
    SIBClass CDATA #IMPLIED
    SIBRevision CDATA #IMPLIED
```

```
    RTCRevision CDATA #IMPLIED
    LCCRevision CDATA #IMPLIED
>
<!ELEMENT SIBActivator (#PCDATA)>
<!ELEMENT Actualparameter (Parametername, Parametervalue)>
<!ELEMENT Parametername (#PCDATA)>
<!ELEMENT Parametervalue (#PCDATA)>
<!ELEMENT SIBDatablock (SIBData+)>
<!ELEMENT SIBData (ProcessedData | ResultData | Errormessage)>
<!ELEMENT ProcessedData (#PCDATA)>
<!ATTLIST ProcessedData
    Datatype CDATA #REQUIRED
    DataId CDATA #REQUIRED
    Revision CDATA #IMPLIED
>
<!ELEMENT Branchexecution (#PCDATA)>
<!ELEMENT ResultData (Returncode, Expectedresult, Obtainedresult)>
<!ATTLIST ResultData
    Datatype CDATA #REQUIRED
    DataId CDATA #REQUIRED
>
<!ELEMENT Returncode (#PCDATA)>
<!ELEMENT Expectedresult (#PCDATA)>
<!ELEMENT Obtainedresult (#PCDATA)>
<!ELEMENT Errormessage (#PCDATA)>
<!ELEMENT Testresult (Resultcode, ExecSummary, ComponentSummary+)>
<!ELEMENT Resultcode (#PCDATA)>
<!ELEMENT ExecSummary (ControlExecutions, TCExecutions)>
<!ELEMENT ControlExecutions (#PCDATA)>
<!ELEMENT TCExecutions (SumOfTestcases, TestcasesPassed,
                                        TestcasesFailed)>
<!ELEMENT SumOfTestcases (#PCDATA)>
<!ELEMENT TestcasesPassed (#PCDATA)>
<!ELEMENT TestcasesFailed (#PCDATA)>
<!ELEMENT ComponentSummary (#PCDATA)>
<!ATTLIST ComponentSummary
    SIBExecutions CDATA #REQUIRED
    ReceivedResponses CDATA #REQUIRED
    UnexpectedResponses CDATA #REQUIRED
>
```

Bibliography

[Abr87] S. Abramsky. Observational Equivalence as a Testing Equivalence. *Theoretical Computer Science*, 53:225–241, 1987.

[ADLU91] A. Aho, A. Dahbura, D. Lee, and M. Uyar. An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours. *IEEE Transactions on Communications*, 39(11):1604–1615, 1991.

[Age] AGEDIS. http://www.agedis.de.

[Ang87] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 2(75):87–106, 1987.

[ANS83] ANSI/IEEE. Glossary of Software Engineering Terminology. ANSI/IEEE Standard 729-1983, 1983.

[Ber94] P. Bernhard. A Reduced Test Suite for Protocol Conformance Testing. *ACM Transactions on Software Engineering and Methodology*, 3(3):201–220, 1994.

[BG92] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BMSY97] V. Braun, T. Margaria, B. Steffen, and H. Yoo. Automatic Error Location for IN Service Definition. In *Proc. of 2nd Int. Workshop on Advanced Intelligent Networks (AIN '07)*, volume 1385 of *Lecture Notes in Computer Science*, pages 222–237. Springer Verlag, 1997.

[Bra01] V. Braun. *A Coarse-granular Approach to Software Development allowing Non-Programmers to Build and Deploy Reliable, Web-based Applications*. Ph.D. thesis, University of Dortmund, Germany, 2001.

[Bri88] E. Brinksma. A Theory for the Derivation of Tests. In S. Aggarwal and K. Sabnani, editors, *Proc. of the Int. Conference on Protocol Specification, Testing and Verification (PSTV VIII)*, pages 63–74. 1988.

[BSS86] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS Specifications, their Implementations and their Tests. In B. Sarikaya and G. Bochmann, editors, *Proc. of the Int. Conference on Protocol Specification, Testing and Verification (PSTV VI)*, pages 349–360. 1986.

[BT00] E. Brinksma and J. Tretmans. Testing Transition Systems: An Annotated Bibliography. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Proc. of Summer School MOVEP'2k Modelling and Verification of Parallel Processes*, pages 44–50. 2000.

[CC92] U. Celikkan and R. Cleaveland. Computing Diagnostic Information for Incorrect Processes. In *Proc. of the Int. Conference on Protocol Specification, Testing and Verification (PSTV XII)*, pages 263–278. 1992.

[CCI89a] CCITT. ISDN User-Network Interface Layer 3 – General Aspects. Rec. Q.930, 1989.

[CCI89b] CCITT. ISDN User-Network Interface Layer 3 – Specification for Basic Call Control. Rec. Q.931, 1989.

[CCI93] CCITT. Typical DSS 1 service indicator codings for ISDN telecommunications services. Rec. Q.939, 1993.

[CH93] R. Cleaveland and M. Hennessy. Testing Equivalence as a Bisimulation Equivalence. *Formal Aspects of Computing*, pages 1–20, 1993.

[Cho78] T. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.

[Cla97] A. Claßen. *Component Integration into METAFrame*. Ph.D. thesis, University of Passau, 1997.

[CS90] R. Cleaveland and B. Steffen. A Preorder for Partial Process Specifications. *Proc. of the Int. Conference on Concurrency Theory (CONCUR '90)*, 458:141–151, 1990.

[CSMB97] A. Claßen, B. Steffen, T. Margaria, and V. Braun. Tool Coordination in METAFrame. Technical Report MIP-9707, University of Passau, 1997.

[CVI89] W. Chan, S. Vuong, and M. Ito. An improved Protocol Test Generation Procedure based on UIOs. In *Proc. of the Int. Symposium on Communication Architectures & Protocols (SIGCOM '89)*, pages 283–294. ACM Press, 1989.

[DAC97] M. Dwyer, G. Avrunin, and J. Corbett. A System of Specification Patterns. http://www.cis.ksu.edu/santos/spec-patterns, 1997.

[DAC98] M. Dwyer, G. Avrunin, and J. Corbett. Property Specification Patterns for Finite-State Verification. In *Proc. of the 2nd Workshop on Formal Methods in Software Practice*, pages 7–15. ACM Press, 1998.

[DAC99] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proc. of the Int. Conference on Software Engineering*, pages 411–420. ACM Press, 1999.

[DN87] R. De Nicola. Extensional Equivalences for Transition Systems. *Acta Informatica*, (24):211–237, 1987.

[DNH84] R. De Nicola and M. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 24:83–133, 1984.

[EJP97] E. Emerson, S. Jha, and D. Peled. Combining Partial Order and Symmetry Reduction. In *Proc. of the 3rd Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 19–34. Springer Verlag, 1997.

[Eme90] E. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of theoretical computer science*. Elsevier, 1990.

[EP90] N. Evtushenko and A. Petrenko. Synthesis of Test Experiments in some Classes of Automata. *Automatic Control and Computer Science*, 24(4):50–55, 1990.

[Eur94] European Computer Manufactures Association (ECMA). Services for Computer Supported Telecommunications Applications (CSTA) Phase II, 1994.

[Eur98] European Computer Manufactures Association (ECMA). Services for Computer Supported Telecommunications Applications (CSTA) Phase III, 1998.

[Eur00] European Computer Manufactures Association (ECMA). Scenarios for Computer Supported Telecommunications Applications (CSTA) Phase III. Technical Report TR/82, 2000.

[Eur03] European Telecommunications Standards Institute (ETSI). TTCN-3. http://www.etsi.org, 2003.

[FHP02] E. Farchi, A. Hartman, and S. Pinter. Using a model-based Test Generator to Test for Standard Conformance. *IBM Systems Journal (Software Testing and Verification)*, 41(1):89–110, 2002.

[FJJ+96] J. Fernandez, C. Jard, T. Jéron, L. Nedelka, and C. Viho. Using On-the-Fly Verification Techniques for the Generation of Test Suites. In R. Alur and T. A. Henzinger, editors, *Proc. of the 8th Int. Conference on Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 348–359. Springer Verlag, 1996.

[FJJ+97] J. Fernandez, C. Jard, T. Jéron, L. Nedelka, and C. Viho. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. *Science of Computer Programming*, 29(1–2):123–146, 1997.

[FvBK+91] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test Selection Based on Finite State Models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gla96] S. Gladstone. *Testing Computer Telephony Systems and Networks*. Telecom Books, 1996.

[Gna] GNATS. http://www.gnu.org/software/gnats.

[God94] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*. Ph.D. thesis, Universite de Liege, 1994.

[God99a] J. Godskesen. Test Generation from Fault Models for Redirected Outputs. In *Proc. of the IEEE Int. High Level Design Validation and Test Workshop (HLDVT '99)*. IEEE Computer Society Press, 1999.

[God99b] J. Godskesen. Two Algorithms for Generating Tests for Embedded Systems. In *Proc. of the 14th Int. Symposium on Computer and Information Sciences (ISCIS '99)*. 1999.

[Gol67] E. Gold. Language Identification in the Limit. *Information and Control*, 10:447–474, 1967.

[Gol72] E. Gold. System Identification via State Characterization. *Automatica*, 8:621–636, 1972.

[Gon70] G. Gonenc. A Method for the Design of Fault-Detection Experiments. *IEEE Transactions on Computing*, 19:551–558, 1970.

[GPY02] A. Groce, D. Peled, and M. Yannakakis. Adaptive Model Checking. In J.-P. Katoen and P. Stevens, editors, *Proc. of the 8th Int. Conference for Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 357–370. Springer Verlag, 2002.

[HCP91] N. Halbwachs, P. Caspi, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. *Anoter Look at Real Time Programming, Proceedings of the IEE, Special Issue*, 1991.

[Hee98] L. Heerink. *Ins and Outs in Refusal Testing*. Ph.D. thesis, University of Twente, The Netherlands, 1998.

[Hen85] M. Hennessy. Acceptance Trees. *Journal of the ACM*, 32(4):896–928, 1985.

[Her] Herakom GmbH. http://www.herakom.de.

[HHM+02a] A. Hagerer, H. Hungar, T. Margaria, O. Niese, B. Steffen, and H. Ide. Demonstration of an Operational Procedure for the Model-Based Testing of CTI Systems. In R. Kutsche and H. Weber, editors, *Proc. of the 5th Int. Conference on Fundamental Approaches to Software Engineering (FASE '02)*, volume 2306 of *Lecture Notes in Computer Science*, pages 336–339. Springer Verlag, 2002.

[HHM+02b] A. Hagerer, H. Hungar, T. Margaria, O. Niese, B. Steffen, and H. Ide. An Operational Procedure for Model-Based Testing of CTI Systems. *Annual Review of Communication*, 56, 2002.

[HHNS02] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model Generation by Moderated Regular Extrapolation. In R. Kutsche and H. Weber, editors, *Proc. of the 5th Int. Conference on Fundamental Approaches to Software Engineering (FASE '02)*, volume 2306 of *Lecture Notes in Computer Science*, pages 80–95. Springer Verlag, 2002.

[HM85] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM*, 32(1):137–161 137–161, 1985.

[HMN+01] A. Hagerer, T. Margaria, O. Niese, B. Steffen, G. Brune, and H. Ide. Efficient Regression Testing of CTI-Systems: Testing a complex Call-Center Solution. In *Annual Review of Communication*, volume 55, pages 1033–1040. Int. Engineering Consortium (IEC), 2001.

[HNS03] H. Hungar, O. Niese, and B. Steffen. Domain-Specific Optimizations in Automata Learning. In W. Hunt and F. Somenzi, editors, *Proc. of the 15th Computer-Aided Verification Conference (CAV '03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer Verlag, 2003.

[Hoa85] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hof97] J. Hofmann. *Program Dependent Abstract Interpretation*. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, Germany, 1997.

[Hol97] A. Holzmann. *Der METAFrame Interpreter: Entwicklung und Implementierung eines dynamischen Modulskonzeptes*. Master's thesis, University of Passau, 1997.

[Hol98] A. Holzmann. Using the PLGraph Library: Developers Guide. Technical report, Chair of Programming Systems, University of Dortmund, Germany, 1998.

[Hol99a] A. Holzmann. The High-Level-Language: Programming language of the METAFrame interpreter. Technical report, Chair of Programming Systems, University of Dortmund, Germany, 1999.

[Hol99b] A. Holzmann. Writing Adapter Specifications. Technical report, Chair of Programming Systems, University of Dortmund, Germany, 1999.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[Hya] Eclipse/Hyades. http://www.eclipse.org.

[ISO84] ISO. Information Processing Systems - Open Systems Interconnection - Basic Reference Model. ISO/IEC 7498, 1984.

[ISO86] ISO. Standard Generalized Markup Language (SGML). ISO 8879, 1986.

[ISO91] ISO. Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework. International Standard IS-9646, 1991.

[ISO96] ISO. Proposed ITU-T Z.500 and Committee Draft on "Formal Methods in Conformance Testing". CD 13245-1, 1996.

[JJKV98] C. Jard, T. Jéron, H. Kahlouche, and C. Viho. Towards Automatic Distribution of Testers for Distributed Conformance Testing. In *Proc. of the Joint Int. Conference on Formal Description Techniques for Distributed System and Communication/Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV '98)*, pages 353–368. Kluwer Academic Publishers, 1998.

[JP89] B. Jonsson and J. Parrow. Deciding Bisimulation Equivalences for a Class of Non-Finite-State Programs. *Symposium on Theoretical Aspects of Computer Science*, pages 421–433, 1989.

[KSK00] S. Kang, J. Shin, and M. Kim. Interoperability Test Suite Derivation for Communication Protocols. *Computer Networks*, 32:347–364, 2000.

[KV94] M. Kearns and U. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[Lan90] R. Langerak. A Testing Theory for LOTOS Using Deadlock Detection. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Proc. of the Int. Conference on Protocol Specification, Testing, and Verification (PSTV IX)*, pages 87–98. 1990.

[LKHH00] C. Liu, D. Kung, P. Hsia, and C. Hsu. Structural Testing of web applications. In *Proc. of the Int. Symposium on Software Relibility Engineering (ISSRE '00)*, pages 84–96. 2000.

[LMS01] B. Lindner, T. Margaria, and B. Steffen. Ein personalisierter Internetdienst für wissenschaftliche Begutachtungsprozesse. In *Elektronische Geschäftsprozesse, Universität Klagenfurt*. 2001.

[LPvB93] G. Luo, A. Petrenko, and G. von Bochmann. Selecting Test Sequences for Partially-Specified Nondeterministic Finite State machines. Technical Report IRO-864, Departement d'IRO, Universite de Montreal, Canada, 1993.

[LT89] N. Lynch and M. Tuttle. An Introduction to Input/Ouput Automata. *CWI Quaterly*, 2(3):219–246, 1989.

[LvBP94] G. Luo, G. von Bochmann, and A. Petrenko. Test Selection based on Communicating Nondeterministic Finite-State Machines using a Generalized Wp-Method. *IEEE Transactions on Software Engineering*, 20:149–162, 1994.

[LY96] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. In *Proc. of the IEEE*, volume 84, pages 1090–1123. 1996.

[Maz87] A. Mazurkiewicz. Trace Theory. In W. e. a. Brauer, editor, *Petri Nets, Applications and Relationship to other Models of Concurrency*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer Verlag, 1987.

[Mer] Mercury Interactive. Winrunner. http://www.mercuryinteractive.com.

[MET] METAFrame Technologies GmbH. http://www.metaframe.de.

[MET99] METAFrame Technologies GmbH. *The Agent Building Center: User's Guide*, 1999.

[Mic] Microsoft Cooperation. http://www.microsoft.com.

[Mic01] Microsoft Cooperation. Using TAPI 2.0 and Windows to create the Next Generation of Computer-Telephony Integration, 2001.

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

[Mil89] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[MNS02a] T. Margaria, O. Niese, and B. Steffen. Demonstration of an Automated Integrated Test Environment for Web-based Applications. In S. Leue, editor, *Proc. of the 9th Int. SPIN Workshop on Model Checking of Software (SPIN '02)*, volume 2318 of *Lecture Notes in Computer Science*, pages 250–253. Springer Verlag, 2002.

[MNS02b] T. Margaria, O. Niese, and B. Steffen. A Practical Approach for the Regression Testing of IP-based Applications. In *IP Applications and Services 2003: A Comprehensive Report*, pages 195–208. Int. Engineering Consortium (IEC), 2002.

[MNSE02] T. Margaria, O. Niese, B. Steffen, and A. Erochok. System Level Testing of Virtual Switch (Re-)Configuration over IP. In *Proc. of the IEEE European Test Workshop (ETW '02)*, pages 67–74. IEEE Computer Society Press, 2002.

[Moo56] E. Moore. Gedanken-experiments on Sequential Machines. *Annals of Mathematics Studies (34), Automata Studies*, pages 129–153, 1956.

[NC95] V. Natarjan and R. Cleaveland. Divergence and Fair Testing. In *Proc. of the 22th Int. Colloquium on Automata, Languages and Programming (ICALP '95)*, number 944 in Lecture Notes in Computer Science, pages 648–659. Springer Verlag, 1995.

[NMH$^+$00] O. Niese, T. Margaria, A. Hagerer, M. Nagelmann, B. Steffen, G. Brune, and H. Ide. An Automated Testing Environment for CTI Systems Using Concepts for Specification and Verification of Workflows. In *Annual Review of Communication*, volume 54, pages 927–935. Int. Engineering Consortium (IEC), 2000.

[NMH$^+$01] O. Niese, T. Margaria, A. Hagerer, B. Steffen, G. Brune, H. Ide, and W. Goerigk. Automated Regression Testing of CTI-Systems. In *Proc. of the IEEE European Test Workshop (ETW '01)*, pages 51–57. IEEE Computer Society Press, 2001.

[NMN$^+$00] O. Niese, T. Margaria, M. Nagelmann, B. Steffen, G. Brune, and H. Ide. An Open Environment for Automated Integrated Testing. In *Proc. of the 4th Int. Conference On Software and Internet Quality Week Europe (QWE '00)*, pages 570–593. 2000.

[NMS02] O. Niese, T. Margaria, and B. Steffen. Automated Functional Testing of Web-Based Applications. In *Proc. of the 5th Int. Conference On Software and Internet Quality Week Europe (QWE '02)*, pages 157–166. 2002.

[NNH$^+$01] O. Niese, M. Nagelmann, A. Hagerer, K. Kolodziejczyk-Strunck, W. Goerigk, A. Erochok, and B. Hammelmann. Demonstration of an Automated Integrated Testing Environment for CTI Systems. In H. Hußmann, editor, *Proc. of the 4th Int. Conference on Fundamental Approaches to Software Engineering (FASE '01)*, volume 2029 of *Lecture Notes in Computer Science*, pages 249–252. Springer Verlag, 2001.

[NSM+01] O. Niese, B. Steffen, T. Margaria, A. Hagerer, G. Brune, and H. Ide. Library-based Design and Consistency Checks of System-level Industrial Test Cases. In H. Hußmann, editor, *Proc. of the 4th Int. Conference on Fundamental Approaches to Software Engineering (FASE '01)*, volume 2029 of *Lecture Notes in Computer Science*, pages 233–248. Springer Verlag, 2001.

[NT81] S. Naito and M. Tsunoyama. Fault Detection for Sequential Machines by Transition-Tours. In *Proc. of the Int. Symposium on Fault Tolerant Computing Systems (FTCS '81)*, pages 238–243. 1981.

[Obj99] Object Management Group. The Common Object Request Broker: Architecture and Specification, Revision 2.3. http://www.corba.org, 1999.

[Par81] D. Park. *Concurrency and Automata for Infinite Strings*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, 1981.

[Pet91] A. Petrenko. Checking Experiments with Protocol Machines. In J. Kroon, J. Heijink, and E. Brinksma, editors, *Proc. of the IFIP 4th Int. Workshop on Protocol Test Systems*, pages 83–94. North-Holland, 1991.

[PvBY96] A. Petrenko, G. von Bochmann, and M. Yao. On Fault Coverage of Tests for Finite State Specifications. *Computer Networks and ISDN Systems (special issue on Protocol Testing)*, 29, 1996.

[PVY99] D. Peled, M. Vardi, and M. Yannakakis. Black Box Checking. In J. Wu, S. T. Chanson, and Q. Gao, editors, *Proc. of the Joint Int. Conference on Formal Description Techniques for Distributed System and Communication/Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV '99)*, pages 225–240. Kluwer Academic Publishers, 1999.

[PW99] R. Probert and A. Williams. Fast Functional Test Generation Using an SDL model. In *Proc. of the 12th Int. Conference on Testing Communicating Systems (IWTCS '99)*. 1999.

[PYvBD96] A. Petrenko, N. Yevtushenko, G. von Bochmann, and R. Dsoulli. Testing in Context: Framework and Test Derivation. Technical Report IRO-1011, Departement d'IRO, Universite de Montreal, Canada, 1996.

[Raf02] H. Raffelt. *Automatisiertes Testen von Internet-Applikationen*. Master's thesis, University of Dortmund, 2002.

[Rat] Rational Inc. The Rational Robot. http://www.rational.com/products/robot.

[RMI] Remote Method Invocation. http://java.sun.com/products/jdk/rmi.

[RNHW98] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic Testing of Reactive Systems. In *Proc. of the 19th IEEE Symposium on Real-Time Systems (RTSS '98)*, pages 200–209. 1998.

[RS98] J. Romijn and J. Springintveld. Exploiting Symmetry in Protocol Testing. In S. Budkowski, A. Cavalli, and E. Najm, editors, *Proc. of the Joint Int. Conference on Formal Description Techniques for Distributed System and Communication/Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV '98)*, pages 337–352. Kluwer Academic Publishers, 1998.

[RT01] F. Ricca and P. Tonella. Building a Tool for the Analysis and Testing of Web Applications: Problems and Solutions. In T. Margaria and W. Yi, editors, *Proc. of the 7th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 372–388. Springer Verlag, 2001.

[SCK+95] B. Steffen, A. Claßen, M. Klein, J. Knoop, and T. Margaria. The Fixpoint Analysis Machine. In J. Lee and S. Smolka, editors, *Proc. of the 6th Int. Conference on Concurrency Theory (CONCUR '95)*, volume 962 of *Lecture Notes in Computer Science*, pages 72–87. Springer Verlag, 1995.

[SD88] K. Sabani and A. Dahbura. A Protocol Testing Procedure. *Computer, Networks and ISDN Systems*, 14(1):285–297, 1988.

[Seg93] R. Segala. Quiecence, Fairness, Testing, and the Notion of Implementation. In E. Best, editor, *Proc. of the 4th Int. Conference on Concurrency Theory (CONCUR '93)*, volume 715 of *Lecture Notes in Computer Science*, pages 324–338. Springer Verlag, 1993.

[Sie] Siemens AG. http://www.siemens.de.

[Sie01] Siemens AG. HiPath AllServe 150 Personal CallManager, Version 1.0, 2001.

[SKGH98] M. Schmitt, B. Koch, J. Grabowski, and D. Hogrefe. Autolink - A Tool for Automatic and Semi-automatic Test Generation from

SDL-Specifications. Technical Report A-98-05, Medical University of Lübeck, Germany, 1998.

[SL88] D. Sidhu and T. Leung. Experience with Test Generation for Real Protocols. In *Proc. ACM SIGCOMM '88*, pages 257–261. 1988.

[SM99] B. Steffen and T. Margaria. METAFrame in Practice: Design of Intelligent Network Services. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 390–415. Springer Verlag, 1999.

[SMB97] B. Steffen, T. Margaria, and V. Braun. The Electronic Tool Integration Platform: Concepts and Design. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 1(1+2), 1997.

[SMBK97] B. Steffen, T. Margaria, V. Braun, and N. Kalt. Hierarchical Service Definition. *Annual Review of Communication*, 51:847–856, 1997.

[SMC+96a] B. Steffen, T. Margaria, A. Claßen, V. Braun, and M. Reitenspieß. An Environment for the Creation of Intelligent Network Services. *Intelligent Networks: IN/AIN Technologies, Operations, Services, and Applications – A Comprehensive Report*, pages 287–300, 1996.

[SMC+96b] B. Steffen, T. Margaria, A. Claßen, V. Braun, M. Reitenspieß, and H. Wendler. Service Creation: Formal Verification and Abstract Views. In *Proc. of the 4th Int. Conference on Intelligent Networks (ICIN '96)*, pages 96–101. 1996.

[SMCB96] B. Steffen, T. Margaria, A. Claßen, and V. Braun. Incremental Formalization: A Key to Industrial Sucess. *Software: Concepts and Tools*, 17(2):78–91, 1996.

[Sti92] C. Stirling. Modal and Temporal Logics. *Handbook of Logic in Computer Science*, pages 477—563, 1992.

[Sun] Sun Microsystems Inc. The JavaBeans Bridge for ActiveX. http://java.sun.com/beans/bridge.

[TB99] J. Tretmans and A. Belifante. Automatic Testing with Formal Methods. In *Proc. of the 7th European Int. Conference on Software Testing, Analysis & Review (EUROSTAR '99)*. 1999.

[Tel] Telelogic. Telelogic Tau.

[Tes] Testing Technologies IST GmbH. TT Series. http://www.testingtech.de.

[The] The Open Group. Test Environment Toolkit. http://tetworks.opengroup.org/.

[TP98] Q. Tan and A. Petrenko. Test Generation for Specifications Modeled by Input/Output Automata. In *Proc. of the 11th IFIP Workshop on Testing of Communicating Systems (IWTCS '98)*, pages 83–99. 1998.

[TPvB96] Q. Tan, A. Petrenko, and G. von Bochmann. Deriving Tests with Fault Coverage for Specifications in the Form of Labeled Transition Systems. Technical Report IRO-1073, Departement d'IRO, Universite de Montreal, Canada, 1996.

[Tre92] J. Tretmans. *A Formal Approach to Conformance Testing*. Ph.D. thesis, University of Twente, The Netherlands, 1992.

[Tre96a] J. Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In T. Margaria and B. Steffen, editors, *Proc. of the 2nd Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer Verlag, 1996.

[Tre96b] J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. Technical Report 96-26, Centre for Telematics and Information Technology, University of Twente, The Netherlands, 1996.

[TVJ00] L. Tanguy, C. Viho, and C. Jard. Synthesizing Coordination Procedures for Distributed Testing of Distributed Systems. In *Proc. of ICDCS Workshop Distributed System Validation and Verification (DSVV '00)*, pages E67–E74. IEEE Computer Society Press, 2000.

[UZ93] H. Ural and K. Zhu. Optimal Length Test Sequence Generation Using Distinguishing Sequences. *IEEE Transactions on Networking*, 1(3):358–371, 1993.

[Val84] L. Valiant. A Theory of the Learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[Val93] A. Valmari. On-The-Fly Verification with Stubborn Sets. In *Proc. of the 5th Int. Conference on Computer Aided Verification (CAV '93)*, volume 697 of *Lecture Notes in Computer Science*, pages 397–408. Springer Verlag, 1993.

[Vas73] M. Vasilevskii. Failure Diagnosis of Automata. *Kibernetika*, 4:98–108, 1973.

[vBP94] G. von Bochmann and A. Petrenko. Protocol Testing: Review of Methods and Relevance for Software Testing. In *Proc. of the Int. Symposium on Software Testing and Analysis (ISSTA '94)*, pages 109–124. ACM Press, 1994.

[vdBBC+97] M. von der Beeck, V. Braun, A. Claßen, A. Dannecker, A. Dannecker, C. Friedrich, D. Koschützki, T. Margaria, F. Schreiber, and B. Steffen. Graphs in METAFrame: The Unifiying Power of Polymophism. In *Proc. of the 3rd Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 112–129. Springer Verlag, 1997.

[VT95] A. Valmari and M. Tienari. Compositional Failure-based Semantic Models for Basic-LOTOS. *Formal Aspects of Computing*, 7(4):440–468, 1995.

[Wal88] D. Walker. Bisimulation and Divergence in CCS. In *Proc. of the 3rd Annual IEEE Symposium on Logic in Computer Science (LICS '88)*, pages 186–192. Computer Society Press, 1988.

[Wora] World Wide Web Consortium. The eXtensible Markup Language (XML). http://www.w3.org/XML.

[Worb] World Wide Web Consortium. The eXtensible Stylesheet Language (XSL). http://www.w3.org/Style/XSL.

[Wor94] World Wide Web Consortium. Uniform Ressource Locator (URL). RFC 1738, 1994.

[Wor99a] World Wide Web Consortium. HyperText Markup Language (HTML). http://www.w3.org/TR/html401, 1999.

[Wor99b] World Wide Web Consortium. Hypertext Transfer Protocol (HTTP). RFC 2616, 1999.

[Wor99c] World Wide Web Consortium. Uniform Resource Identifier (URI). RFC 2396, 1999.

[YCL98] N. Yevtushenko, A. Cavalli, and L. Lima. Test Suite Minimization for Testing in Context. In *Proc. of the IFIP 11th Int. Workshop on Testing of Communicating Systems (IWTCS '98)*, pages 127–145. 1998.

[Yoo03] H. Yoo. *Fehlerdiagnose beim Model Checking durch animierte Strategie-Synthese*. Ph.D. thesis, University of Dortmund, 2003.