University of Newcastle upon Tyne
Computing Science Department

Automated Test Generation from Algebraic Specifications

Ph.D. Thesis

by
Andrej Pietschker

August 2001

Paginated blank pages are scanned as found in original thesis

No information is missing

Abstract

This thesis is a contribution to work on the specification-based testing of computing systems. The development of computing systems is a challenging task. A great deal of research has been directed at support for analysis, design and implementation aspects, yielding a wide range of development techniques. However, the crucial area of system testing remains relatively under-explored.

Because a project may spend a good part of its budget on testing, even modest improvements to the cost-effectiveness of testing represent substantial improvements in project budgets. Relatively little literature has been devoted to the entire testing process, including specification, generation, execution and validation. Most of the academic literature seems to assume a revolutionary change of the testing framework. On the contrary industry follows a more traditional approach consisting of trusted methods and based on personal experience. There is a need for testing methods that improve the effectiveness of testing but do so at reasonable cost and which do not require a revolutionary change in the development technology.

The novel goal of the work described in this thesis is to "lift" traditional testing so that it takes advantage of system specifications. We provide a framework – *hep*TEST– which is motivated by this goal. To that end, *hep*TEST is a framework consisting of a specification language, a technology for generating tests in accordance with test strategies, a means of applying the tests to the implementations and support for validation of outcomes against the specification-based tests.

We will first categorise different testing methodologies and then examine some of the past and present approaches to test data: we develop only the necessary theoretical foundations for *hep*SPEC and always consider the requirements of testing. The formalism *hep*SPEC for system description is based upon a well-defined algebraic approach. It utilises a novel approach allowing the description of finite domains in a way suitable for engineering purposes. The engineers' tasks are to provide an adequate description of the system in

*hep*SPEC.

The approach proposed in this thesis is grounded in the traditional approach to testing where test data is provided to the system under test and the outcome is compared to the expected outcome. To enhance the capabilities of the framework a general order on test inputs is proposed to be used in test strategies. Traditional testing strategies requiring an order on test inputs are introduced and their realisation in *hep*TEST discussed as well as a proposal of new strategies which lend themselves to this particular approach.

The manipulation of the specification yields abstract test cases which are then transformed into test cases suitable for the chosen implementation of the system. This transformation, called test reification, is necessary to bridge the "abstraction gap" between the abstract specification-derived tests and the concrete implementation on which the test must run. The transformation is necessary in order for the approach to be practical and is achieved through homomorphisms which are expressed in specially adapted grammars. This transformation is also applied to the generated test outcome and is aimed there at easing test result validation.

The utility of the *hep*TEST approach is illustrated by means of a simple example, a larger case study and one carried out within the aviation industry.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr John Fitzgerald for his constant support and constructive advice. I am grateful to Dr Fitzgerald for his comments and criticisms on the preliminary drafts of the work. Special thanks are extended to Dr Jason Steggles and Dr Maciej Koutny of the thesis committee, for their useful suggestions and especially Dr Steggles for his support as my supervisor during the write-up period.

I am grateful to Professor Marie-Claude Gaudel and Dr Bruno Marre from the "Laboratoire de recherche en informatique" at the University of Paris Sud for the fruitful discussions and Dr Marre for his patient and continuous help with LoFT.

I would like to thank Tom Brookes of BAE SYSTEMS Avionics and BAE SYSTEMS Dependable Computing Systems Centre for the greatly appreciated guidance and assistance in the industrial case study.

I would like also to thank several of my colleagues and staff members of the Centre for Software Reliability for their prompt help and technical support on many occasions and for being such a lovely hosts for the time I send in Newcastle. These include Professor Tom Anderson, Dr Jim Armstrong, Joan Atkinson, Dr Oliver Biberstein, Dr Joseph Chu, Professor John Dobson, Neil Henderson, Dr Steve Riddle, Dr Amer Saeed, Claire Smith, and Dr Ros Strens. Also, many thanks to Shirley Craig for her patience and efficient help in searching out many relevant references for this thesis.

The support and encouragement offered by my parents and my wife during studies are also greatly acknowledged.

Contents

List of Figures

List of Examples

Chapter 1

Introduction

This thesis concerns the role of testing in the development of computing systems. In particular, it addresses the use of system specifications to enhance the process of test derivation, application and analysis. Its particular contribution is in lifting test development and application procedures to take advantage of formal system specifications. This chapter provides the initial motivation for our work. A brief discussion of the industrial costs and benefits of testing leads to the identification of the main aim for the research reported here. The structure of the remainder of the thesis is presented.

Industry is looking to constantly improve the quality of its products. This demand for quality is driven by the competitive nature of the market and, for example in critical or high-integrity systems, by strict industry standards. The drivers for quality are also sometimes at odds with the desire to drive down development costs. Testing is an important factor in the quality assurance process.

We may describe testing simply as an activity consisting of the execution of an executable on known inputs and the comparison of the visible behaviour with an expected outcome. In this sense testing has many facets. For example the validation of executable models of a system where the comparison is simply with expectation can be testing, but also the traditional approach where testing is used for system verification where the comparison is with a specification. A closer look at the different aspects of testing is provided at the beginning of Chapter 2.

System or requirements testing is seen as a means of increasing the confidence that a software product or system accurately realizes customer requirements. In safety-critical systems, the level of assurance required is particularly stringent. Yet, although proof of correctness is seen as one means of attaining this, testing is still of importance, especially

where COTS components and standard operating systems are used. There are high costs associated with both proof and testing, hence substantial research in the automation of both activities. Since testing exercises the software developed, rather than subjecting it to a static analysis as is the case with proof, it is unlikely that proof will replace testing to any extent in the medium term.

Testing is facing a number of problems. Testing is a labour intensive task. Test cases have to be designed and implemented, executed and the test results validated. Many commercial products support automated test execution, but test case creation is rarely automated. The difficulty in test case creation stems from two sources. First exhaustive testing is impossible. Dijkstra's (E.W. Dijkstra, 1972, p. 6) famous comment is usually quoted to support the claim that "*Program testing can be used to show the presence of bugs, but never to show their absence!*" This deep insight into the dilemma of testing has a big impact on test case creation. In the case of automated test creation it means that not all possible tests could be created. It forces the tester to select test cases which are likely to expose faults or where successful tests would increase the confidence into the correctness of the implementation. But to find those relevant tests manually is difficult. Tedious labour is required to create tests which follow a selected strategy. The complexity of problems causes complex solutions which in turn require complex analysis to select the relevant tests.

The other source of problems in test creation is the validity of created tests. When an executed test case fails, i.e. it did not produce the expected result, then there are three possible causes:

- *Implementation is incorrect.* This can be verified by analysing the possible causes. The usually technique is debugging and source code inspection.
- *Specification is incorrect.* The specification is either contradictory or does not reflect customer requirements correctly. This needs to be established by analysing the specification, e.g., by inspection.
- *Test is incorrect.* The test case and the system specification are contradictory. This can be verified by comparing the test case to the system specification.

The situation is problematic because all of these analysis techniques are very laborious and cost intensive. The first cause can not be eliminated. It is inherent to the testing process that the presence of bugs is discovered. The second source should have been eliminated much earlier in the development process. However, it is difficult to point to a test failure

source instantly. Therefore it is highly desirable to eliminate possible test failure sources. Our desire in the work described in this thesis is to eliminate the third cause, i.e. the possibility to create invalid tests with respect to the specification. The solution can be seen in a tight coupling of specification and test case creation.

The general aim of the work described here is to exploit specifications in order to yield practical improvements in the test selection, application and outcome analysis.

Specification-based testing (Poston, 1996) aims to exploit the system specification in order to generate suitable tests for the implementation. Confidence into the correctness will be gained from successful tests.

A number of problems must be faced in automatic test generation from system specifications:

- *Suitable framework.* The selection of the framework within which the tests will be generated plays a vital role in the success of the automation attempt. Some frameworks do not cater for developer's needs as the expressiveness is limited, others are unsuitable for automation (Donat, 1997). Other problems are specific to the underlying theory or chosen approach and vary widely, see e.g. (Dick and Faivre, 1993) or (Carrington and Stocks, 1994). A detailed discussion of these approaches follows in Chapter 2.3.

- *Test selection.* There may be an infinitely large space of tests from which a choice must be made. This makes automation difficult and calls for additional assumptions that may not be satisfied in the given context. In order to make test selection viable, a *test strategy*, which defines the relevance of tests, must be proposed. Test strategies are developed for company wide use or can be specific to an application domain. Some testing methods impose a selection strategy on the user. Testers are very reluctant to adopt other strategies mainly because they have a strong believe into those which were developed for their needs.

- *Abstract vs. Concrete Tests.* The use of specifications introduces an "abstraction gap" between the specification-derived tests and the implementation. The result is that specification-based tests cannot be applied directly to the implementation because the representation of abstract test values is in general very different from that needed to drive the implementation. This results in an additional transformation step which should be closely tied to the formalism used to derive tests. This would limit the possibility of generating invalid tests. It is important for any specification-

based approach which aims to generate tests to find a method to bridge this gap automatically without too much effort.

- *Test Result Validation.* Test result validation in the traditional approach compares the actual outcome to the expected outcome. When tests are created manually, the tester usually provides the expected result, which he derives from his knowledge of the problem domain. In an automated setting the generation of an expected outcome is vital to the practicality of the approach. Otherwise the tester is forced to derive the result manually and the benefits of automatic generation are reduced. This also has to influence the decision about a suitable framework, because it must allow us to compute the outcome. This is not possible in all frameworks. Additionally we should require that the expected outcome is made available in a representation which eases test result validation. What has been said previously about abstract vs. concrete tests applies also to the expected outcome.

The specific objectives of the work reported in this thesis are:

1. To develop a framework for specification-based testing that is designed with the developer's needs in mind, and is suitable for automation and exploits rigour in the specification language.

2. To show how such a framework can aid test selection and to provide foundational and practical support for such selection.

3. To provide support within the framework for bridging the abstraction gap between abstract and concrete tests.

4. To provide automated support within the framework for result validation.

Given our aim of providing a developer-centred framework for specification based testing, we propose to adapt traditional, industrially proven techniques (Beizer, 1995) and aim to achieve a high degree of automation to improve the cost-effectiveness and reliability of tests. The traditional approach is "lifted" to the abstract level of the specification, where we can use formal techniques to specify the properties of the desired system. These properties are automatically transformed into test cases. This frees testers from the tedious tasks of selecting and creating test cases manually. Other proposals for automated test case generation tend to impose a testing strategy on the user. In the spirit of traditional testing techniques the selection of appropriate test cases, also known as the testing strategy, is

user-driven in our proposal. The user, or tester in this case, has the task to select the most appropriate test strategy. He does this using his knowledge about design, implementation techniques used in this project and his experience in identifying relevant test cases. Our main goal is to support him in his efforts through a high degree of automation.

This thesis will first introduce some testing techniques and examine past and present proposals for test case generation. Then we reason about our proposal and introduce the alternatives and discuss them. The next chapter contains the formal basis of our specification language and the next introduces our testing approach. We exemplify these new techniques in two case studies, where one is an industrial project. The thesis concludes with an assessment of the techniques and proposes for future research.

Chapter 2

Test Case Generation

This chapter provides a classification of testing, an overview of functional testing technology and a discussion of previous approaches to automatic test data generation.

2.1 Testing Techniques

Testing has been an important part of the development cycle for decades. Various claims have been made about testing, but it can be seen as a means of gaining confidence into the correctness of the system. As correctness has many meanings so testing has many faces. In order to place the work reported in this thesis into context, we distinguish six:

- *Functional Testing.* Tests concentrate on establishing that the functionality of a system has been implemented correctly. These tests usually disregard performance issues and related requirements. The emphasis lies on correctness. A test oracle compares expected and actual outcome and yields success or failure. Functional testing, in particular, testing w.r.t. a specification, is discussed in greater depth in Section 2.2.

- *Regression Testing.* Modification of parts of existing systems can have a bigger impact than desired. Regression tests are performed to confirm that the changes have not introduced errors into parts of the system formerly considered correct.

- *Conformance Testing.* Some systems are required to conform to standards. Tests are produced according to the requirements of the given standard. Test success is described in or is derived from the standard in question.

- *Usability Testing.* This kind of testing is aimed at testing the Human-Computer-Interface. Commonly used in the presence of graphical user interfaces, tests are performed manually by independent testers and success is assessed according to factors like ease of use, and system guidance.

- *Documentation Testing.* Documentation is an important part of the system and requires testing too. Among the goals of documentation testing are establishing documentation's correctness, completeness, and usability.

- *Load Testing.* Typical client/server-applications or volume-critical systems, such as databases, require tests exercising the transfer rates of the system. Satisfaction is rated according to the rates achieved or the volume processed.

Despite the different focuses some tasks are common to all these testing approaches - the general tasks of test selection, test production, application of tests and test assessment.

First appropriate tests have to be generated. This task consists of test selection and test generation. These two tasks are usually coordinated. Some approaches generate tests and select the appropriate ones from this set. This method is very expensive, as useless tests are produced at length, so it is in practice only applicable where generation is automated (Offutt, 1988). Usually tests will be produced so that they satisfy a selection criterion. A test selection criterion determines which tests are appropriate. The test generation process is stopped when the criterion is satisfied. The criterion is what varies from testing approach to testing approach. Figure 2.1 presents a variety of criteria which might apply.

Tests can be selected so that they cover the source code structurally. Well-known structural coverage criteria are path and statement coverage (Beizer, 1990). In cases where a specification is used the testing criterion can be based upon the structure of it. Alternatively the criterion can be based upon a model which is derived from the specification and describes the functionality or aspects of it. We talk then about specification or functional coverage. Tests can also be based upon examination of the input domains and partitions.

But such a selection criterion is not always apparent, not talking about formally defined, even though it indisputably exists. In the case of usability testing, for example, the user decides on an ad-hoc basis which tests he selects (Beizer, 1990). The selection criterion is implicit whereas other testing approaches have a well defined test selection criterion, such as "cover all statements of the source code".

Next the tests have to be applied to the system under test. Running the tests means loading the system, providing an initial known state and supplying the system with test in-

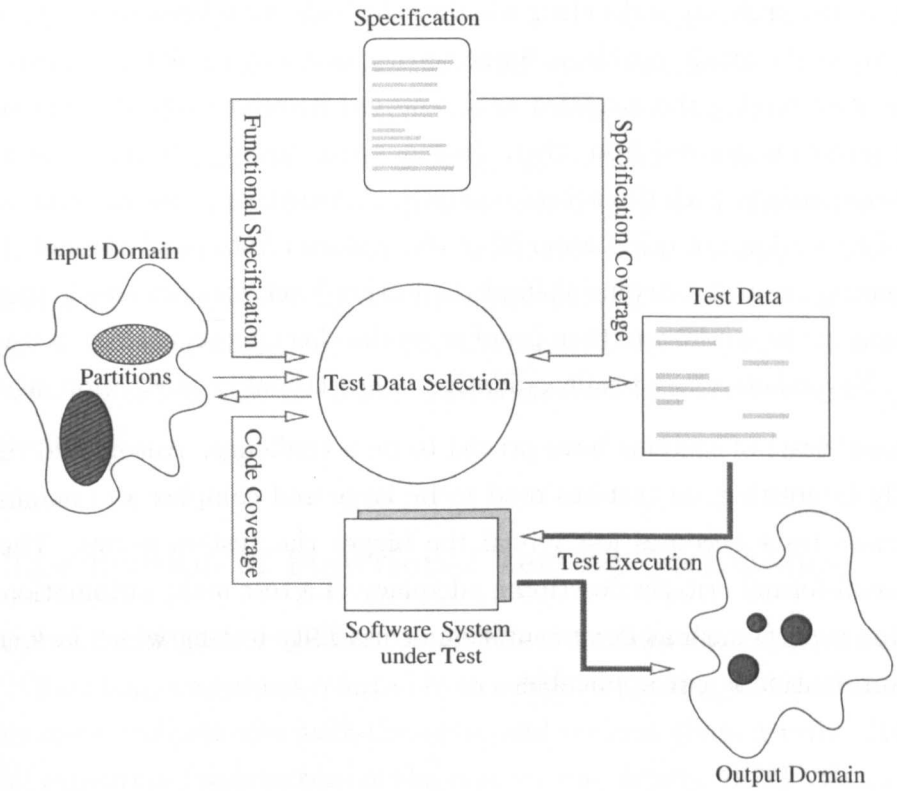


Figure 2.1: Test Data Selection in the Testing Process

puts. These steps are performed automatically, semi-automatically or in general manually. Costs, and the intervals at which these tests are run, dictate the choice.

The results of the tests have to be assessed to determine success or failure of a test. A test is said to have been successful if it produced an outcome in accordance with expectations, otherwise it failed. A test failure identifies the presence of a bug. Test result assessment might be done by comparison with an expected result or by evaluation of the outcome. The evaluation can be machine assisted or can be performed manually. There is in either way the problem of deciding whether the tests have been successful or failed, this is referred to as the oracle problem. Some testing methods resolve this problem easily, for example in load testing the achieved and expected transfer rates are compared, and test success or failure is derived from that. In functional testing the outcome has to be predicted for comparison with the actual outcome. Alternatively the outcome of the test can be assessed by evaluation using a model of the system (Antoy and Hamlet, 1992). Rather then comparing the test outcome at implementation level, the outcome is transformed into its equivalent in the model and then proof or model checking technology is used to evaluate the result. Nevertheless test result validation is not trivial in most approaches.

Functional tests of systems have proved to be a challenge. Automated test generation is especially interesting, as systems tend to be large and complex and manual production of appropriate tests becomes less trivial the bigger the system grows. The existence of formal or semi-formal criteria describing adequacy of a test make automation feasible. For other testing aspects such as documentation or usability testing which lack such a definite criterion automation is not applicable.

2.2 Functional Testing

Functional testing is discussed in great detail by Myers (1979), and again by Beizer (1990). The famous remark by E.W. Dijkstra (1972, page 6): "Program testing can be used to show the presence of bugs, but never to show their absence!" has been used as an argument against testing. The key idea in functional testing is, however, that systematic test selection can increase confidence in the correctness of an implementation rather than provide complete assurance.

2.2.1 Approaches to Correctness Assurance

Even in the light of system development with formal methods testing will continue to play a leading role. Modern systems are constructed using tools, libraries, compilers, and run on operating systems and utilise hardware. Such system structures form a pyramid and the specifications cover only the peak of it, not the entire structure. If validation techniques would be required to show the correctness of the implementation, then these techniques need to be applied to the entire structure. Because of the complexity of modern systems, the development environment where many libraries are Components-Of-The-Shelf (COTS), such validations would be immensely resource and time consuming, if to be successful at all. In this sense formal methods of verification or validation such as proof cannot ensure correctness of the implementation without being impractical. Those results are applicable only to the models or the level of abstraction on which they where carried out. Proof can increase our confidence into the correctness of our specification, testing into the correctness of the implementation. Therefore we see these methods as complementary. The goals of *hep*TEST, the testing framework developed in this thesis, are concerned with the verification of systems. We believe that test execution can demonstrate that the system is operationally correct implemented.

2.2.2 Classification of Functional Testing Techniques

Functional testing techniques are differentiated into two main streams, glass-box and black-box testing. Glass-box, sometimes referred to as white-box testing, is based on source code. This includes static analysis, like walk-throughs, and revision (Sommerville, 2001) and dynamic testing exploiting the structure of the source code (Myers, 1979) . Static techniques have their own advantages and disadvantages, but in system testing they are less applicable than during unit or module testing (Beizer, 1990). The black-box testing approach makes no use of the source code but uses a different source of description of the implementation. The black-box approaches differ in the way test cases are selected, and we can distinguish between nondeterministic and deterministic testing. In the nondeterministic case tests are selected either randomly or according to a given heuristic. Each time tests are produced a different test set is obtained, therefore the name nondeterministic. Dyer (1992) discusses random testing in detail, where heuristics can differ and are usually based on the use of operations during the lifetime of the implementation. To find faults in the implementation tests would be chosen emphasising operations that are rarely used in practice, e.g.

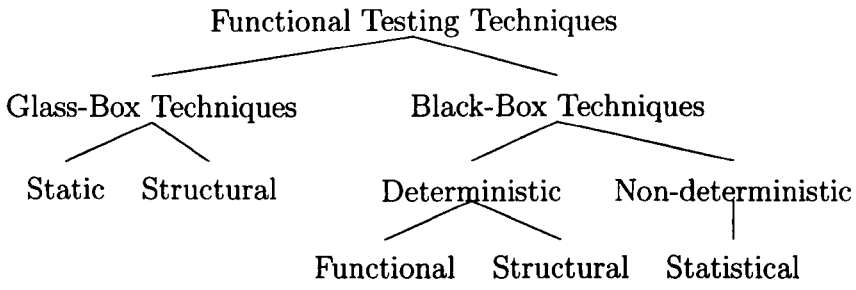


Figure 2.2: Testing Approaches to Functional Testing

setup, initialisation and maintenance. Other heuristics may favour operations that will be used frequently. In contrast the deterministic approach selects test cases according to a criterion based on a system's description. This description could be a formal or informal specification. The selection criterion is then based upon the structure or the functionality of the system. The selection criterion gives rise to the same test set every time tests are produced, hence the name deterministic. Figure 2.2 presents an overall picture of testing approaches. Up to now we discussed glass-box and black-box testing techniques in general, but they need to be accessed to their use in system testing.

2.2.3 White-Box Testing Approach

The white-box testing approach makes use of the source code to produce tests. The coverage criteria vary from "all paths" to "all statements". Most of these techniques are based upon the structure of the source code. The "all statements" criterion requires a test set ensuring that all statements in the source code are executed at least once during the execution of the tests.

In the light of modern development techniques, like client/server architecture or event-driven programming satisfying the selection criteria becomes more of a challenge. The source code does not contain enough information about the order of execution, in other words the structure of the code does not reflect the systems architecture. In event-driven programming, for example, the code consists of small pieces of linear code. The functionality is hidden in the operating system referencing these code pieces. An attempt to find a path in the code may fail if no knowledge about the operating system and its behaviour is present.

Research by Strooper and Hoffman (1991) into test data generation for C modules using Prolog has been aimed at finding typical mistakes. The main problem in using the

source code is that the notion of the statements can only be guessed. Mistakes resulting from omission cannot be detected by white-box testing techniques. The insufficiency of the white-box approach in dealing with system tests has been shown by others (Myers, 1979; Poston, 1996). The main reason is that there is no independent source of the systems expected behaviour in white-box testing.

To document the requirements independently from the implementation a specification can be written. Using this document rather than the implementation leads to black-box testing, where the only source of information used during system testing is its specification. The name black-box testing stems from the notion that the internal states and behaviour of the system is hidden and only its interface is visible to the tester.

2.2.4 Specification-Based Black-Box Approach

Black-box testing does not necessarily require a formal specification. Approaches dealing with heuristics (Ince, 1987) or genetic algorithms (Michael et al., 1997) can generate test inputs automatically. However, because they do not produce expected outcomes, test result validation is a problem. The inability to validate test results nearly rules out these approaches for functional testing. They are applicable to other approaches, where test result validation does not require an expected outcome, like load testing, or can be used if a reference system exists, that can produce the expected outcome for test validation.

The specification-based black-box approach can be applied conveniently to systems which can be abstracted to a function relating inputs to outputs. A formal specification describing this function is needed if tests are to be generated automatically (Donat, 1997). We will have to choose one or a combination of theories within which the properties of the function are stated. The earlier mistakes are discovered the easier they are to fix and therefore the costs of system development are kept low.

This is the reason why the objectives of testing emphasise early test data generation: functional tests can be produced right after the requirements have been recorded. If these requirements are recorded using a formal method then, among other benefits, automatic test case generation is possible. Figure 2.3 provides an overview of how specification-based automatic test data generation is integrated into the development process.

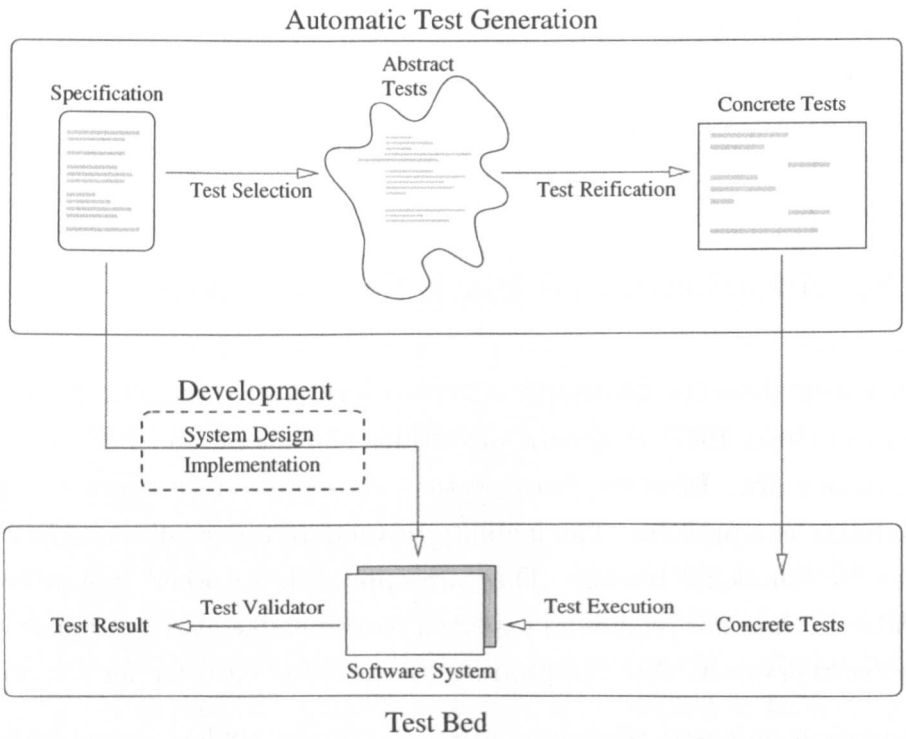


Figure 2.3: Integration of Test Generation into the Specification Based Testing Approach

2.3 Test Generation

This is an overview of existing test generation approaches according to their specification method. The various approaches differ so much in the way they generate tests, what is generated or what is used as a basis that we have assessed them using some very general criteria reflecting the goals of automatic test case generation. Evaluation of each approach is based on the following criteria:

- *Expressiveness* of the specification method. This determines the extent to which the method is widely applicable or weather it is only suited to a specific application area.
- *Automation level* achievable. Because of the underlying theory there are obstacles which might hinder the achievement of a higher level of automation.
- *Test selection* support. From a practical point of view the method should support various test selection methods. This allows the engineer to select the appropriate one.
- *Test reification* implemented. We understand test reification as the process of transforming abstract tests derived from a specification into concrete tests understood by the system. This point is closely related to the one about automation level. If test reification is not supported by the method then the level will be lower. Nevertheless test reification is very important and can be achieved in various ways. This criterion should assess the way test reification is handled.

2.3.1 Finite State Machine

Finite state machines are well understood and supported. Test generation from descriptions of finite state machines (Fujiwara et al., 1991), extended finite state machines (Cheng and Krishnakumar, 1993), X-machines (Ipate and Holcombe, 1998) has been subject of research for many years. The approaches cover many aspects of test generation from finite state machines, such as the use of Unique-Input-Output sequences for test validation, and optimisation of test sets.

Finite state machines and closely related approaches are successfully employed to describe protocols, hardware and such systems. They are not suitable to describe an entire software/hardware system if the expressive power of finite states machine is exceeded. Due to the size of the systems targeted by this work it has to be concluded that finite state

machine based approaches are insufficient regarding their expressiveness and abstraction. They also lack the ability to deal with complex data structures in general and have to be turned down as a basis.

2.3.2 Grammars

Another widely understood formalism is that of context-free grammars. They are more expressive than finite state machines and can therefore be applied to a greater variety of problems. Grammars in general have become the formalism in compiler construction. So unsurprisingly the first work on test data generation from grammars has been targeting test generation for compiler tests (Purdom, 1972). Many algorithms have been developed for structural coverage. But it has emerged that the algorithms were optimised for the size of the test rather than for exposing faults (von Mayrhauser et al., 1994a).

Continued work by Bauer and Finger in (Bauer and Finger, 1979) suggested great potential. Others followed on like Lindquist and Jenkins in (Lindquist and Jenkins, 1988). And in the 1990's the work was taken up by Maurer in (Maurer, 1990) and Burgess in (Burgess, 1993).

These approaches have been mainly limited to parsers and similar systems. The general drawback was expressiveness. Using attributed grammars this was overcome but the development and maintenance has become too difficult (von Mayrhauser et al., 1994a).

2.3.3 Model-oriented Specification

Model-oriented specification languages such as Z and VDM-SL have been successfully adopted in the industry. Since the early 1990's these formalisms have been a source of investigations for test generation.

Stocks and Carrington have produced a number of results on test generation from Z specifications (Stocks and Carrington, 1991, 1993a,b, 1996). They used a formal description of the tests in Z to produce test frames. These test frames contain the information of how tests are to be produced. A disadvantage is that the description of the system is not used but the source is an additional formal specification, describing the tests themselves. That means that maybe two specifications have to be maintained and held consistent. This disadvantage can turn into an advantage if no formal description of the system exists, because tests can be generated from an independent specification. The special circumstances in a specific task will decide what is true.

Dick and Faivre used VDM specification of systems to generate tests for single functions (Dick and Faivre, 1993). A series of problems was overcome in this research. The main cause of concern results from the expressive power of model-oriented formalisms. The use of implicit function definition leads to problems when tests are sequenced and outcomes computed. Partitioning of input domains is also affected. The approach proposed by Dick and Faivre (1993) constructs a finite state machine relating preconditions to postconditions. It became apparent that the construction of these machines will require user interference.

Hörcher and Peleska suggested the use of Z as a basis (Hörcher and Peleska, 1995). They overcome the disadvantage of the approach by Stocks and Carrington (1996) by using the specification of the system. But they too cannot overcome the problems with implicit function definitions.

Donat (1997) generalised all these approaches considering propositional logic with quantification. He showed that the level of automation achievable is limited by the formalism. The use of quantifiers allows to generate logical schemata that specify groups of black-box test cases, not individual test cases.

2.3.4 Algebraic Specification

Algebraic specifications are another formalism which has been considered by researchers for test generation. Early work started in 1986 by Bougé, Choquet, Fribourg, and Gaudel as described in (Bougé et al., 1986). This work has been continued over the years by Bernot, Gaudel, and Marre in (Bernot et al., 1991a; Gaudel, 1995) tackling a variety of problems.

Researchers at the "Laboratoire de recherche en informatique" described how to generate tests automatically from algebraic specifications in 1986 (Bougé, Choquet, Fribourg, and Gaudel, 1986) and influenced the work reported in this thesis. Figure 2.4 summarises their general testing approach.

Testing in the sense of the paper by Bougé et al. (1986) means answering the question: "Does implementation X satisfy axiom Y?" A test case is then a ground equation, where the variables in the axiom Y are substituted by ground terms. Automation is achieved through tools finding these substitutions. Systems capable of finding such substitutions are logical term rewriting systems such as Prolog. There is a connection between algebraic specifications and logic programming exploited to produce prototype implementations of a specification (Bouma and Walters, 1989). The same idea is used by Bougé et al. to convert the specification into a logical program. To generate tests for a particular axiom Y the process is as follows. The transformation of axiom Y into a Prolog goal is the initial step.

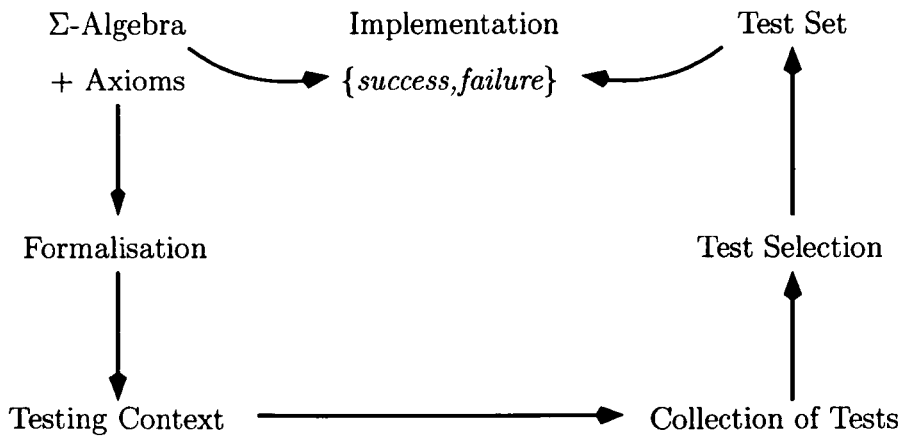


Figure 2.4: Test Generation and Selection

The resolution algorithm of Prolog will search for all possible solutions. After test case selection the relevant tests are applied to the implementation X and failure or success is determined by an oracle.

The first step is to transform the specification into a logical program. Then the axiom under test is transformed into a goal. The Prolog resolution algorithm returns all possible substitutions for the unbounded variables in the goal.

From these, in general infinitely many cases, some tests are selected according to hypotheses.

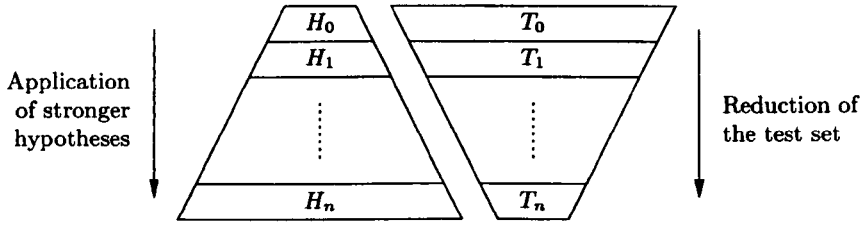


Figure 2.5: Test Hypothesis

The idea is to start with weak hypotheses resulting in a huge, maybe even infinite, test set. By strengthening the hypotheses the test set size is reduced becoming of practical interest as illustrated in Figure 2.5. The hypotheses may range from an initial hypothesis, stating the finitely generated condition on the implementation, ranging to the hypothesis, that the program is correct. In the latter case there is no need to test and the size of the test set becomes zero. If hypotheses can be assured, say by proof, then the combination

of testing and proof ensures correctness of the implementation w.r.t. the specification. In practice this may be too expensive or simply impossible for reasons discussed earlier, so the choice of hypotheses is crucial to the test effort and its result.

Bougé et al. suggest to use the regularity hypothesis. This hypothesis requires a metric of complexity for each sort and assumes, that if tests with inputs of complexity up to k are successful, then all inputs would be valid. The engineer would provide a metric for each sort, such as the term's depth. This can be difficult from time to time, so another hypothesis called uniformity hypothesis is derived from the assumption, that if one test from a domain is successful, than all will be.

Hypotheses should be chosen in a way, such that possible, likely faults will be uncovered. The hypotheses proposed by Bougé et al. are too strong. Traditional testing techniques require far weaker assumptions. Beizer (1995), driven by practical concerns, suggests for example to select test covering boundary values and we will follow his ideas.

In the theory of algebraic specification we can derive other equations than those which are explicitly stated. In the proposal of Bougé et al. these equations are not included in the test generation. Thus these properties are not tested.

And there are other problems resulting from the idea of testing with the hypothesis that implementations are finitely generated algebras as each specification may contain auxiliary sorts or operators. For an implementation to be a finitely generated algebra would mean that all operators from the specification have to be implemented and become a part of it. Auxiliary operators and sorts are not an integral part of the problem description, but are introduced for convenience, for example to state a property. Such auxiliary operators or sorts will find no expression in the implementation. The generated tests cannot be executed. To solve this problem Bougé et al. *require* that the implementation is a finitely generated algebra. This is a restriction which prohibits the use of auxiliary operators or sorts or requires their existence in the implementation.

Summarising, it has to be questioned if testing by answering the question "Does the axiom hold for the implementation?" has any meaning. If we follow the ideas of Bougé et al. then we need to ensure that the system has implemented the derivation process correctly. This is a hypothesis which is not supported by any evidence. Moreover it is difficult to believe. The author is unaware of systems which do implement such logic features.

The tests derived by Bougé et al. (1986) exercise the system and determine if it does what it should. Such tests we will call *positive tests*. The suggestions by Beizer (1995) lead

to *negative tests*. Negative tests determine if a system has no additional properties or if it does not do what it should not. In the case of algebraic specifications we can use initiality to answer such questions. If an algebra is an initial model of a specification then only their axioms and those which can be derived hold. In order to generate negative tests we need to assume that the intention was to develop a system which behaves like an initial model.

Woodward criticises the work of Bougé et al. (1986) in his paper (Woodward, 1993). He argues that the generated test cases lack the link to the implementation. Understandably he is suspicious if a formal term or equation is accepted as input by the implementation. This is a problem that needs to be investigated.

2.3.5 Object Oriented Models

Object oriented models have been the source of recent investigations. With the widespread use of object oriented technology during analysis, design and implementation phases special interest has grown to use these models during testing. The aim is to reduce the time needed to develop testing model and reuse existing documents during testing phases.

For example von Mayrhauser, Mraz, and Walls in (von Mayrhauser et al., 1994a,b) describe a system called Sleuth. There they use an object model and enrich the specification with pre and post-conditions and add a finite state machine system which describes the interaction between the commands. This approach contains difficulties. Firstly the specification is constructed using 3 different formalisms. There is no unifying theory which would allow to use the results from one theory in the others. This brings up the question if the generated tests are valid. Secondly the specification is very close to the implementation. The lack of abstraction capabilities makes it very difficult to specify an entire system. Moreover the specification does not describe a system but a specific installation of it. Thus tests are answering the question: "Would this installation function correctly?" and focuses on the installation and not the system. The difficulty is that the new tests have to be created for each new installation. However the abstraction gap we mentioned in the discussion about algebraic methods is not present here, because the abstraction is minimal. This raises the question if these are black-box or white-box testing techniques.

Poston (1994) reports a similar approach. He also has to overcome the lack of semantics present in an OMT model. He also uses pre and post-conditions and additionally annotates the model with domain information. The result are test cases which do not contain an expected outcome. The reason is probably the same as in the case of model-oriented specifications and due to the expressive power of pre- and post-conditions. It is argued that

the expected outcome is unnecessary and not desired because otherwise the specification could replace the implementation. This is a difficult argument to believe, because firstly an executable specification is probably too slow to be a replacement for program code. Secondly it is too difficult to deem it practical that a huge number of automatically generated tests has to be assessed manually. This includes analysis, computations and checks for a huge number of values. Even for trivial applications this seems to be very inefficient.

A similar assessment is provided by Liggesmeyer and Rüppel (1996) where the lack of clear semantics of object-oriented models is criticised. The authors also argue that testing object-oriented programs requires new techniques because structural coverage criteria are not easily applicable to object-oriented software.

2.4 Summary

The variety of existing and newly developed selection criteria calls for support during test case generation if testing should be practical. The choice of the selection criteria reflects the testers experience in finding faults. Testers are reluctant to accept methods imposed upon them by tools. Automation of test generation yields systematically developed test cases.

We aim to generate system tests utilising the power of a formal system to lift traditional approaches to testing to a more formal level.

Our first goal is to find a suitable formalism which we can use to describe our system under test (SUT). It is important to us that the specification can be used to automatically generate black-box tests for the SUT.

We have surveyed a number of formalisms to look at their advantages and disadvantages. The degree of automation achievable is determined to a certain degree by the choice of the underlying theory. Therefore this choice is crucial.

Following our selection criteria we have excluded those formalisms which do not possess the expressiveness to describe a system in full. The use of techniques based upon finite state machines has been ruled out for this approach on the basis that the expressive power is limited. Similar things have to be said about grammars, despite the possibility to increase expressiveness by using attributed grammars, because software engineering considerations reveal drawbacks. This practically leaves the choice between model-oriented and algebraic formalisms. Unfortunately model-oriented specifications have to be dismissed too, this time because their expressiveness limits the degree of automation achievable.

The popularity of object-oriented methods could have made them a good choice, unfortunately the models lack clear semantics and cannot be used to calculate the outcome.

Algebraic specifications and their use for test case generation are discussed in greater detail in Section 3.5. The entire Chapter 3 is devoted to algebraic specifications beginning with an introduction to the theory, discussing problems related to testing from algebraic specifications and proposing a framework for automated test generation.

Surprisingly, none of the visited methods had implemented test reification. Test reification is the link between the abstract model and the systems implementation. Although some methods allow for a level of abstraction none of these methods provided a solution of how the generated abstract tests are mapped to concrete tests. This will be a main goal for this work.

Chapter 3

The Rationale behind *hep*TEST

This chapter provides the motivation for the work undertaken by this thesis. In this chapter we will start by summarising our initial position.

We will motivate the choice of algebraic specification in Section 3.2 and 3.3 and provide an introduction of the concepts of algebraic specification which are then pinned down formally in Chapter 4. We will review the traditional testing approach which has been applied widely in industry in Section 3.4. We will explain in Section 3.5 how our approach proposes to deal with a variety of problems and how our algebraic specifications can be used to generate automatically test cases for functional system test. In Chapter 5 we will go into details which are omitted in this motivating introduction and in Chapter 6 we will apply the proposed technology to a choice of three case studies.

3.1 Goals of *hep*TEST

The driver behind this work – *hep*TEST – is to assist testers in testing the functionality of a system.

In order to test the functionality of a system a tester must create test cases, apply them, and validate the outcome. Before tests can be created a description of the functionality is required. Myers (1979) showed in the well-known triangle example that the implementation cannot be used as a description of the functionality for the purpose of testing.

Myers Triangle Example

Myers (1979) describes in his book an example to show that white-box testing alone cannot find all the bugs in a software. He proposes the task of identifying triangles to illustrate his claim. A software is expected to classify triangles into isosceles, equilateral or scalene. He discusses a possible implementation and creates tests using white-box and black-box techniques. He continues to show that white-box techniques fail to establish the incorrectness of the implementation. He shows that omission as a source for faults cannot be identified by white-box techniques.

In Appendix A we provide a formal specification of the triangle example in our proposed formalism *hep*SPEC.

In order to generate test data automatically we need to have a description which can be manipulated by algorithms in the same sense used by Donat (1997). Donat argues successfully that the formal description is needed so that it can be transformed algorithmically into a set of, as he calls them, test frames. We agree with this observation and say that a formal system for describing the functionality and automated manipulation is required in order to automatically generate test data. Donat (1998) has chosen the formalism of VDM-SL for his system requirements descriptions. So his approach is hampered by the inability to derive test sequences automatically and that is why he generates only test frames which consist of a precondition for the test, a list of stimuli and an expected response. These test frames need to be instantiated, e.g. variables substituted with actual test inputs, such that the chosen test inputs satisfy the conditions provided. Due to the expressiveness of model-oriented specifications this task cannot be automated in general. There are also other approaches using a model-oriented framework (e.g. Dick and Faivre, 1993; Hörcher and Peleska, 1995). As discussed in Chapter 2.3.3 they have problems in generating test sequences which cannot be overcome in general. Test sequences play a major role in system tests. To test a particular subfunction of the system under test a sequence of test data needs to be submitted to the system in order to set a state from which the subfunction in question can be accessed. Test sequences provide the means of reaching the subfunction through the systems interface. Test sequences are not vital for tests in unit testing where the tester has access to the functions state through either direct interference

or by auxiliary procedures. They are usually removed when a system is created and their absence makes the execution of sequences so important to the system test. For *hep*TEST we have chosen to use algebraic specifications following the results in this research area (Bougé et al., 1986; Marre, 1995). Algebraic specifications are suitable for deriving test sequences and to describe systems at an abstract level. This will help the tester to handle the descriptions even of larger systems. In the remainder of this work we will see from the results that the choice of algebraic specification was justified.

The objectives of this approach to testing as outlined in Chapter 1 are a high degree of automation, the use of system descriptions as a basis for test generation and practical viability of the approach. In our search for solutions we will keep a close eye on practical aspects of the work. These include, besides applicability to a wide range of problems and software engineering methodology, the possibility to adapt this approach in different test environments.

In contrast to the work of Bougé et al. (1986) we do not want to prove the correctness of the implementation, our goal is to generate test cases systematically and automatically. The specification is used to derive appropriate tests according to a user given testing strategy. There is no empirical evidence about the quality of such generated test cases, this work wants to establish the foundations and make further research possible.

Assisting the tester in automated test case execution is another goal of *hep*TEST. Automated test execution in conjunction with automated test generation could overcome the problems of frequent changes in the specification during the development process. The changes would be recorded and the description of the system adjusted accordingly. Tests can be regenerated and executed automatically, freeing the user from these tasks.

The third major task in testing is test result validation. In this area *hep*TEST provides support by generating an expected outcome. In order to verify test results the generation of an expected outcome is considered important and becomes another goal in *hep*TEST. Especially in a setting of automated test case generation providing an expected outcome is crucial for the applicability of the framework. It could be argued that there is little value in generated tests without a corresponding outcome or some other possibility to validate the test result like e.g. in Antoy and Hamlet (1992).

Summarising, the goals of *hep*TEST are:

- automated generation
- of functional tests

- focusing on the degree of automation
- and the practical applicability of the method.

This chapter is an in-depth discussion of the alternatives and choices made to achieve the objectives. We will give an introduction to algebraic specifications in so far as is needed to show the choices available and to explain the reasons for the selections made when choosing the theory underlying the method. A critical review of the traditional testing approach as presented by Beizer (1995) is followed by the proposal of *hep*SPEC (*h*ierarchical *e*quational *p*artial *spec*ification) and *hep*TEST, an automatic test case generation framework which addresses some of the deficiencies identified in earlier work. A comparison with earlier work on test generation by Bougé et al. (1986) concludes this chapter.

3.2 Algebraic Specification

Before we can discuss the choice of the algebraic framework we have to introduce informally some of the basic features of algebraic specifications. The precise definitions of *hep*SPEC, the proposed specification language, will be given in Chapter 4. Here we need only some basics to drive the discussion about the rational behind *hep*TEST. We will follow the classical works on algebraic specifications (e.g. Ehrig and Mahr, 1985).

Signature

Let us give a simple example to clarify some terms. We will specify natural numbers and Booleans. There is the obvious need to name the set of natural numbers, here we call them `Nat` and for the set of Boolean values we choose `Bool`. In general such names are called *sorts*. We follow the idea of specifying natural numbers by *zero* and *successor*. To do so we select two operators `zero` and `succ`. With the operators we associate *domains*, sequences of sorts, and *ranges*, a single sort. In our example the operator `zero` has the empty set as its domain and `Nat` as its range. We call operators with an empty domain *constant operators*. Constant operators in our example are also `true` and `false` each having `Bool` as its range. The operator `succ` is unary and has `Nat` as its domain as well as range. The domain and range associated with an operator are called the operator's *arity*. Example 3.1 shows the complete specification. The tuple of sets of sorts, operators and their arity make up a *signature*. A *specification* is a signature presented in the way as in Example 3.1 containing all the keywords.

```
Example1 is
  sorts Nat, Bool
  oprn
     zero: → Nat
     succ: Nat → Nat
     true: → Bool
    false: → Bool
end Example1
```

Example 3.1: Algebraic Specification `Example1` of Natural Numbers and Boolean

We can build *terms* with the syntactical material presented so far in the usual way. The arity of an operator determines the number of arguments, and the sorts of the arguments. With the specification from Example 3.1 we can form, for example, the terms `zero` or `succ(succ(zero))`. We say a term is *of sort* S to mean that the range of the outermost operator of the term is of sort S. When we construct terms, we need to ensure that subterms and domains are of the appropriate sort. For example `succ(zero)` is a valid term, because `zero`'s range is `Nat` and the first argument of `succ` has to be of sort `Nat`. The terms `zero(succ(zero))` or `zero(succ)` are invalid, because they disobey the sort condition. Terms can include variables as place holders for subterms. We call terms without variables *ground terms*.

Enhancement

In practice a project, even a small one, can easily contain 200 sorts, and 500 operators. To handle such a big number of entities we need to provide construction operations which build signatures from signatures. In this way we can structure our description and provide better readability and accessibility. This principle has been discussed in software engineering for years (Wirth, 1971) and has become a standard way to address this issue.

In the algebraic specification theory larger descriptions are constructed by *enhancement* of existing specifications. Let us use Example 3.1 again to clarify this issue. We want to enhance this signature with two new operators `add` and `leq`. Now we need to specify how to add two natural numbers. This is done by *axioms*. An axiom can be a simple *equation*, where right-hand-side and left-hand-side are terms and are separated by the equality symbol. Example 3.2 is the complete specification.

In line 5 we expressed the well-known property $0 + b = b$. The axiom in line 6 states that in order to calculate $a' + b$ we can first calculate $a + b$ and then build the successor of

```
1   Example2 is Example1 with
2     oprn add: Nat Nat → Nat
3     leq: Nat Nat → Bool
4     axioms a,b: Nat
5       add(zero,b) = b
6       add(succ(a),b) = succ(add(a,b))
7       leq(zero,b) = true
8       leq(succ(a),zero) = false
9       leq(succ(a),succ(b)) = leq(a,b)
10  end Example2
```

Example 3.2: Algebraic Specification **Example1** enriched by Less-or-equal and Addition

the result, if a' is the successor of a. In other words we defined add recursively. We took the same approach to define `leq`, defining `leq` first for the constant operators and then through recursion.

Parameterisation

In order to build up a large library of ready-to-use signatures we introduce the concept of parameters. When writing specifications, we realize that certain parts recur (This is the same observation which is utilised in object oriented languages like C++ through the use of templates). In algebraic approaches this is realised by defining parameters for signatures. These parameters can then be substituted by another signature.

Semantics of Algebraic Specifications

Whereas sorts and operations are the syntactical material, axioms provide the meaning of the specification.

An *algebra* is understood as a collection of sets (the so-called carrier sets) and operations. We assign each sort from our specification a set from the algebra and each operator an operation. Then we call an algebra a *model* of the specification if all axioms from the specification hold for all assignments in the algebra.

For example the algebra of natural numbers and Booleans with the usual operations *successor*, *addition*, and *less-or-equal* is a model for our specification in Example 3.2. This is not surprising, because it was our intention to specify natural numbers with addition as well as Booleans.

We can say that the meaning of a specification is its class of models. The axioms

restrict the class of algebras which are models and that is how they fulfil their semantic task.

But we want to restrict the models even further. There is a class of models (called *initial* models) which is of particular interest to us. For an initial model of a signature it is true that every value is represented by at least one term. In other words we can name any value from the algebra by referring to a term. This is a very important property for test generation. It means that we can generate an abstract input for any given test.

Other models include loose and final (or terminal) semantics. Loose semantics can be summarised by saying that a model needs to comply with the stated properties. In general loose semantics are used when we want to assert a property (Goguen and Diaconescu, 1994) whereas initial semantics are used for defining a structure. Final semantics are more complicated than initial and need usually more axioms to describe the system (Wirsing, 1990). Furthermore, initial models have exactly those properties which are stated or can be "computed" from the specification. This means that we do not have properties in our system which we are unaware of and we can also test this using so-called negative tests.

In Chapter 4 we present the chosen formalism and our own restrictions where we explain the necessity of initial models for the generation of negative tests. Before that we will go into the different approaches in handling partiality of functions in algebraic specifications. In the algebras as presented so far the functions are *total*, defined for all values of a carrier set. In practice this is insufficient as described in the following section.

3.3 Algebraic Methods for Handling Partiality

A major drawback of the classical approach to algebraic specifications as presented so far and in e.g. Ehrig and Mahr (1985) is the lack of sophisticated methods for domain construction. Although useful in many ways for the task at hand, total algebras are insufficient because many operators in our daily experience work on restricted domains. This is a result of the fact that we are dealing always with finite resources. We needed to find a way of dealing with this problem and look therefore into the various methods for addressing it.

Total Algebra with Error Values

One approach is to use error values to model partiality (Ehrig and Mahr, 1985). Through axioms values outside the domain are mapped to error values. Using the existing specifica-

tion from Example 3.2 we add the operator sub and the axioms describing the properties of subtraction. The outcome is the specification in Example 3.3. Line 5 expresses the well-known fact that $a - 0 = a$. Line 6 tells us how to calculate sub recursively. In Line 7 we deal with the problem that the first argument might be smaller then the second. In this case we denote this with a special value, an error value, or bottom element $\perp$. This augmentation is partly illustrated in Example 3.3.

```
1     Example3 is Example2 with
2       oprn sub: Nat Nat → Nat
3
4       axioms a,b: Nat
5         sub(a,zero) = a
6         sub(succ(a),succ(b)) = sub(a,b)
7         sub(zero,succ(b)) = ⊥
8         ⋮
9     end Example3
```

Example 3.3: Algebraic Specification Example2 enriched by Subtraction using Error Values

It is obvious that we would have to add axioms about addition and subtraction with bottom elements. This may become a problem when the specification is written. In Example 3.3 we used an existing specification as a basis and included the operator sub. The definition of the operator sub added a value to the sort Nat, the bottom element $\perp$. And this requires axioms defining how the operator add deals with encounters of $\perp$. This is not very difficult in this example, we can use the following axioms:

```
add(⊥,b) = ⊥
add(zero,⊥) = ⊥
add(succ(a),⊥) = ⊥
```

There is more to be done when we consider the impact upon leq. Is the error value comparable? Do we need to introduce a new error value? In theory not, but in practice we need to know exactly what went wrong and error values ought to be distinguishable. Keeping in mind that we will deal with large specifications we might find this difficult. The software engineering principle of stepwise construction as proposed by Wirth (1971) is jeopardised. Each time a specification is enriched the previous definitions need to the reviewed to ensure completeness with respect to handling the error values. In Chapter 4, after introducing some necessary definitions, we will provide a formal proof which highlights the problem.

There are other reasons why we refrain from using error values. In Example 3.3 the partiality of subtraction becomes only apparent from the study of the axioms and our knowledge about the intentional use of the bottom element $\perp$. Only studying the axioms will reveal that the first argument needs to be greater than or equal to the second. The use of error values hides the limits it models from the user in axioms. Bearing in mind that we will deal here with very large system descriptions, this can become an issue to software engineering practice.

For test case generation the use of bottom elements is difficult too. The error value is for the algorithm indistinguishable from any other value of the domain as both cases are mixed together (Wirsing, 1990). The directed test generation for negative testing, as introduced in Section 2.3.4 is therefore impossible. The tester applies negative testing to verify that the system does not do what it should not. A distinction between correct and incorrect behaviour is therefore needed. To overcome this problem other approaches using order sorted algebras have been developed.

Order-Sorted Algebra

Another approach to deal with restricted domains has been developed by Goguen and Diaconescu (1994). Order-sorted algebras can be use to model partiality. Example 3.4 illustrates how sub-sorts can be used to model restricted domains. In this case we introduced the operator `div` which takes only elements of sort `NZ-Nat` as arguments. With the supplied definitions of `zero` and `succ` we can deduce that the intention is to avoid a division by 0. We can compare this approach very well with notations known from mathematics.

```
1   Example3* is
2     sorts Nat, NZ-Nat
3     subsort NZ-Nat < Nat
4     oprn
5       zero: → Nat
6       succ: Nat → NZ-Nat
7       div: Nat NZ-Nat → Nat
8   end Example3*
```

Example 3.4: Order-sorted Approach to Specifying Division

There abbreviations like $\mathbb{R}^+$ for positive or $\mathbb{R}^* = \mathbb{R} \setminus \{0\}$ are commonly used to describe subsets of real numbers.

In computing, however, we often have to deal with restricted intervals and have diffi-

culties describing them using sub-sorts. The required restriction for the operator sub from Example 3.3 cannot be defined by simple sub-sorting. Goguen and Diaconescu (1994) introduce some work on using boolean conditions to describe domain conditions. This might be a more practical approach. A drawback for test case generation, as we propose it, is the existence of polymorphism. The OSA approach is especially designed to deal with polymorphism. It means that one operator symbol may have different arities. In the context of test case generation this is not a desired property, because during test generation it might be unclear which kind of data need to be generated. A good example is the operator add which is frequently used to denote the addition between naturals, integers, reals and other kind of numbers. When we specify the operator using polymorphism we might not care what the actual parameters are when it is used within a particular case, but in test case generation we ought to know this. It would even influence the test selection process, because the boundaries of those sorts are different. So we refrained from using OSA for our purposes.

Operators with Restricted Domains

Wirsing (1990) and Goguen and Diaconescu (1994) point to a third method which provides a formal approach to partiality first presented by Kaphengst and Reichel (1971) and then even more detailed by Reichel (1987). In this approach partiality is modelled by operations with restricted domains. A set of equations is associated with each operator and the solution of this set is the domain of the operation. Returning to the example with operator sub we need to ensure that the first argument is less or equal to the second one. Example 3.5 illustrates how the operator sub could be specified using the idea of domain restriction as proposed by Reichel (1987). The operator `leq` appears in the domain condition for sub. In line 3 the domain of sub is restricted by the equation in line 4 following the keyword `iff`. It means that, if and only if the domain condition holds, the operation can be applied. The fact that subtraction on natural numbers is defined only for those pairs (a, b), for which $b \leq a$ holds, is modelled in a natural and direct way. The partiality of subtraction is not hidden in the axioms but stated in the definition.

In order to define subtraction correctly we have to introduce *conditional equations*. A conditional equation has a *premise* and a *conclusion*. A set of equations constitutes the premise and the conclusion is a simple equation. We use the notation `if` *premise* `then` *conclusion*. For convenience we will omit empty premises and the keywords `if` and `then` and write just the conclusion. So simple equations are a special case of conditional

```
1     Example3' is Example2 with
2     oprn leq: Nat Nat → Bool
3          sub: a:Nat b:Nat → Nat
4              iff leq(b,a) = true
5
6     axioms a,b: Nat
7         leq(zero,b) = true
8         leq(succ(a),zero) = false
9         leq(succ(a),succ(b)) = leq(a,b)
10
11        if leq(zero,a) = true then sub(a,zero) = a
12        if leq(succ(b),succ(a)) = true
13          then sub(succ(a),succ(b)) = sub(a,b)
14    end Example3'
```

Example 3.5: Algebraic Specification of Subtraction using Operations with Restricted Domain Conditions

equations. In Example 3.5 we write the axiom in line 11 using a conditional equation. The equation `leq(zero,a) = true` is the premise and `sub(a,zero) = a` the conclusion. The premise ensures that the operator **sub** is not called outside its defining condition. The specification of total functions does not require the use of conditional equations (Ehrig and Mahr, 1985). In the context of operations with restricted domains, as they are used here, they are essential for the construction of correct specifications. Note, that although the modelled function *sub* is partial, the operation **sub** is total, however, on a restricted domain.

The theory as presented by Reichel (1987) is far too general for our purposes. Our main goal is not the construction of a most general theory for partial algebras but the automated generation of system tests. Reichel (1987) presents a theory which admits specifications for which we cannot generate tests automatically. We are only interested in those specifications which have an initial model. Furthermore we required an expected outcome for each test. This constrains the acceptable range of specifications even further. In order to achieve this we decided to restrict the operations to primitive recursion because they are all computable (Péter, 1967). This is a restriction which other authors (Antoy, 1989; Kaphengst, 1981) consider appropriate because engineers rarely use other kinds of operations. We will follow this suggestion and should this restriction prove to be too restrictive then we might consider weakening it. All of these considerations lead us to the theory which forms *hep*SPEC. The theory is presented formally in Chapter 4.

Using the description we can generate tests. But the number of tests which could be generated is usually very large despite the fact that we consider finite domains. Test selection strategies can be used to selected tests which are considered particularly significant. It is not trivial to decide which tests are significant for an arbitrary system. Here the experience of a tester and his or her knowledge about the system are a valuable asset. In our approach — *hep*TEST— we aim to utilise this asset and to use test selection strategies found in common literature about traditional testing(Myers, 1979; Beizer, 1995). Before we can apply traditional techniques we need to lift them to the abstract level of *hep*SPEC. The next section discusses critically the traditional approach to test selection and introduces the ideas of *hep*TEST.

3.4 Traditional Domain Testing Strategies

The goal of traditional domain testing as described by Beizer (1995) is to find faults in the domain classification process. It is mainly employed in numerical computation. The general technique according to Beizer exploits the idea that the program is described by a classification process and a corresponding treatment, processing, of the input. The classification can be understood as a case-statement over, in the case of Example 3.6, numerical inputs. Afterwards the inputs are processed according to the classification.

The tests are selected through boundary analysis. The case statement is analysed and the boundaries of the input space identified. Two kinds of boundaries can be distinguished, open and closed ones. This terminology is brought from mathematics where open and closed boundaries are formally defined. We will give here a rather informal account of this notion. If a boundary is inclusive, then we call it closed, open otherwise.

The aim of boundary testing is to ensure that the case statement has been implemented correctly. Therefore, for a closed boundary the value just on, the ON-point and the value just off the boundary, the OFF-point, are test inputs of particular interest. For an open boundary the case is just reverse, the value closest to the boundary and the value on the boundary are of interest with the boundary going just between them.

Some assumptions are made:

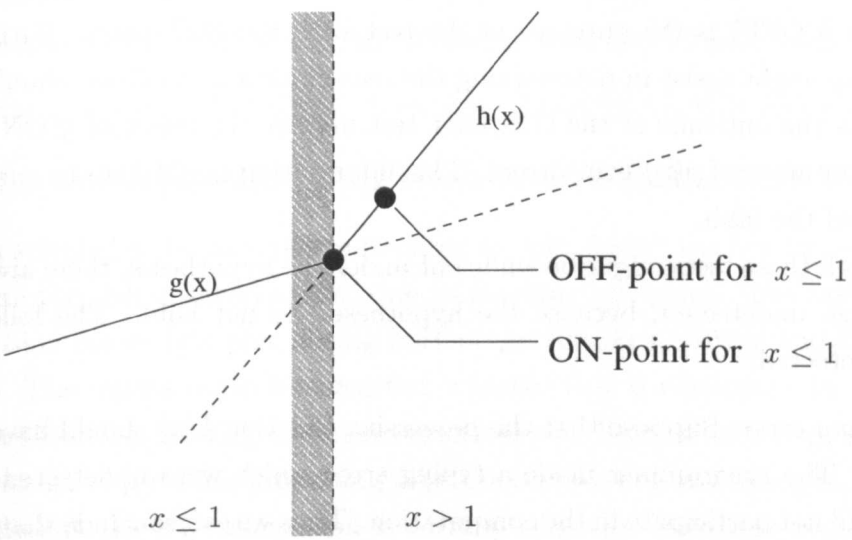
- Coincidental correctness does not occur, which means that if the classification process is incorrect, we can observe an incorrect output. This is justified, because the usual mistakes include missing terms, mistyped terms, misplaced code, wrong constants. Beizer on other faults: "That's no bug, that's sabotage (Beizer, 1995, p. 161)".

- Adjacent domains require different processing. In other words if they do not require different processing then there is no need to distinguish those domains. Domains for which this is true require further investigation, there might be a mistake in the specification.

- The input space is continuous and extends to $+\infty$ and $-\infty$.

- The border between domains is produced by simple predicates. The borders are linear.

$$f : \mathbb{R} \to \mathbb{R};\ f(x) = \begin{cases} g(x) = 5(x-1)^2 + 2 & \text{if } x \leq 1, \\ h(x) = 20(x-1)^2 + 2 & \text{if } x > 1. \end{cases}$$



Example 3.6: Traditional Domain Testing

Example 3.6 is provided to illustrate the technique and to clarify the underlying assumptions. For simplicity let us call the processing function in the first case $g(x)$ and the function in the second case $h(x)$. The domain of f is split by the inequality predicate $x \leq 1$ into two subdomains, $]-\infty, 1]$ and $]1, +\infty[$ respectively. The test strategy requires us to select one test value on the boundary and one off the boundary. The two tests in this case would be 1 for the ON-point and $1 + \epsilon$ for the OFF-point, because the border of the first domain is closed. The same values would be selected for the other domain, thus it is enough to test just these values. This is an result of the improved boundary analysis

technique as presented by Bingchaing Jeng (1994). The tests will be evaluated according to Table 3.1.

Test Result	Result of ON	Result of OFF
Success	$g(\text{ON})$	$h(\text{OFF})$
Border shift to the right	$h(\text{ON})$	$h(\text{OFF})$
Border shift to the left	$\overline{g(\text{ON})}$	$g(\text{OFF})$
Computation g incorrect	$\overline{g(\text{ON})}, \overline{h(\text{ON})}$	$h(\text{OFF})$
Computation f incorrect	$g(\text{ON})$	$\overline{g(\text{OFF})}, \overline{h(\text{OFF})}$
Computation incorrect and Border shift	$\overline{g(\text{ON})}, \overline{h(\text{ON})}$	$\overline{g(OFF)}, \overline{h(\text{OFF})}$

Table 3.1: Error Detection by Domain Testing

With the help of the table we can determine the test result using the actual outcome. For example the test would have been successful if the outcome of the ON-point is equal to $g(\text{ON})$ and $h(\text{OFF})$ is the outcome of the test with the OFF-point. Furthermore the actual outcome might assist in determining the reason for a test failure, should one occur. For example is the outcome of the ON-point test neither the result of $g(\text{ON})$ nor $h(\text{ON})$ then the computation of $g(x)$ is incorrect. This information might help to guide the tester to the source of the fault.

Even though these tests might be sufficient under the hypotheses, there are some errors which might go undetected, because the hypotheses do not hold. The following errors might go undetected:

- *Processing error.* Suppose that the processing function $h(x)$ should have been $2(x-1)^2+2$. The programmer made a typing error which went undetected because the factor did not participate in the computation. Thus we can conclude that coincidental correctness can not be excluded, the initial assumption does not hold. Some mistakes are prone to lead to coincidental correctness, and are difficult to reveal by using only inputs at or close to the boundaries.

- *Classification error.* If an inappropriate ϵ is chosen then an error in the computation might lead to the assumption that a test has failed. If we select ϵ to be the machine precision, then the following computation shows

$$g(OFF) = g(1+\epsilon) = 5(1+\epsilon-1)^2+2 \approx 2 \approx 2(1+\epsilon-1)^2+2 = h(1+\epsilon) = h(OFF)$$

that we would according to Table 3.1 establish a test failure. A correct implementa-

tion would be rejected, a result which is as undesirable as an incorrect implementation being accepted. Thus the selection of ϵ matters in practice and is not trivial as suggested by Beizer. It is even platform dependent so test cases have to be produced for a specific environment. Implementation running on multiple platforms would require different tests. This can result in high testing costs.

Furthermore the assumption that domains are continuous does not hold. In our examples in previous sections we used discrete and non-numerical domains, like Booleans. Even real numbers, as implemented on computers, are discrete. The assumption of continuity lead to a test case which failed on a correct implementation. All computers as of now have upper bounds on number representations. It is wrong to assume that the domains can extend to infinity. An upper or lower bound would give rise to more tests. These can be crucial as computations can produce overflow or underflow and lead to bugs which might be difficult to detect otherwise.

Considering what has been said, the following tests should have been chosen:

- For the interval $[-\text{MAXREAL}, 1]$ choose {-MAXREAL,MIN,AVERAGE,1,1+μ}
- For the interval $]1, +\text{MAXREAL}]$ choose {1+μ,AVERAGE,MAX,MAXREAL}

where μ is selected to be as small as possible to "pin down" the border, but large enough to eliminate the risk of incorrectness due to machine precisions. The problems with machine precision comes into play during both steps, the classification and the processing of the input. The inputs could be classified wrongly if μ is eliminated in the calculations for the classification, leading to an incorrect output, because of wrong processing. But also elimination of μ during processing might occur, leading the tester to believe in a test failure, although the implementation is correct. Therefore selecting μ can be rather difficult. The values -MAXREAL and MAXREAL are included to analyse the stability of the implementation answering the question: "Are these values properly rejected?" The values MIN and MAX are designed to test the computations on the border of the valid input domain. They have to be selected assuring that under-/overflow and other numerical problems are not present. In general MIN and MAX will not coincide with -MAXREAL and MAXREAL. The function as defined in Example 3.6 could not be implemented using the standard numerical expression of programming languages like C. A numerical analysis would show that during computation of f(MAXREAL) an overflow would occur. So the corrected input domain is a subset which would have to be determined for each problem. We need to consider average values from the domains as well, in order to have a higher

confidence in the correctness of the process implementation and we need to consider also n-dimensional non-numerical inputs.

There is a restriction to linear boundaries. The domains have to be numerical and the boundaries are described by linear predicates including relations like greater-than or less-or-equal. We experience many situations where this assumption does not hold.

Nevertheless traditional testing has proven over the years of service in industry that it does have its value. We will show how the disadvantages can be overcome by abstracting from the numerical domain and how they can be lifted to the level of abstraction as provided by *hep*TEST.

In the following section we will provide an outlook on Chapter 5, in which we will summarise our proposals for *hep*TEST in a more informal manner.

3.5 Objectives of *hep*TEST

The chosen approach follows the basic ideas proposed by Bougé et al. (1986), concentrating on overcoming the disadvantages and extending the methodology by lifting traditional testing techniques to the level of abstract specifications as discussed in the previous section.

Let us start by summarising the black-box functional testing approach. When testing a product's functionality we have to execute the implementation, to provide inputs, to register the outcome and to assess test success or failure.

In Figure 3.1 we present the traditional testing process at the bottom of the figure and our idea of using the abstraction of a formal specification in connection with the traditional approach. The specification has to describe the system under test adequately. We then propose to use the specification to automatically derive test inputs. Using those inputs and the specification we can compute the expected outcome for each test. In Section 3.3 we discussed the restrictions, like primitive recursion, which we choose to impose on *hep*SPEC in order to ensure that the computation can yield a result.

Producing a pair of input and expected outcome at the abstract level is insufficient from a practical point of view. To use these inputs in testing the concrete implementation of the system we need to transform the test inputs from the abstract level of the specification to the concrete level of the implementation. This transformation we will call test reification. The reified tests are then applied to the implementation and the outcome is recorded. This leaves as a final step test result validation. Here there are two possible approaches: the comparison can be done either at the abstract or at the concrete levels. In the first case,

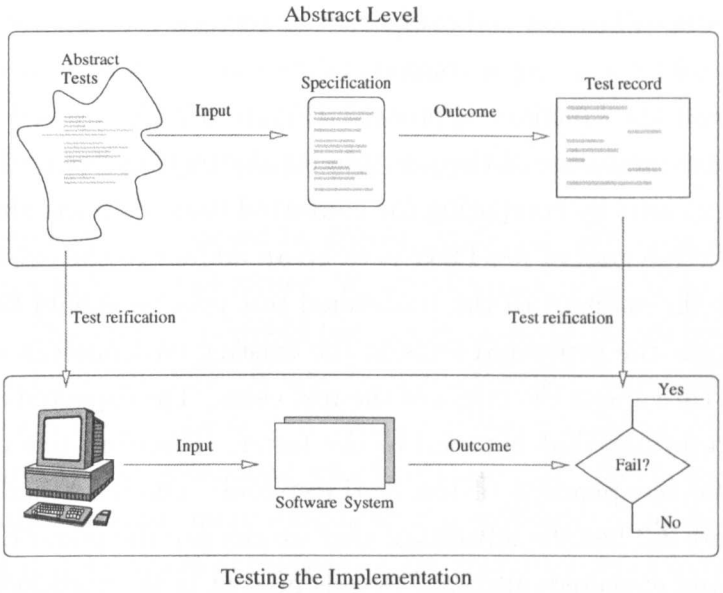


Figure 3.1: Traditional Testing Process Augmented by an Abstract System Specification

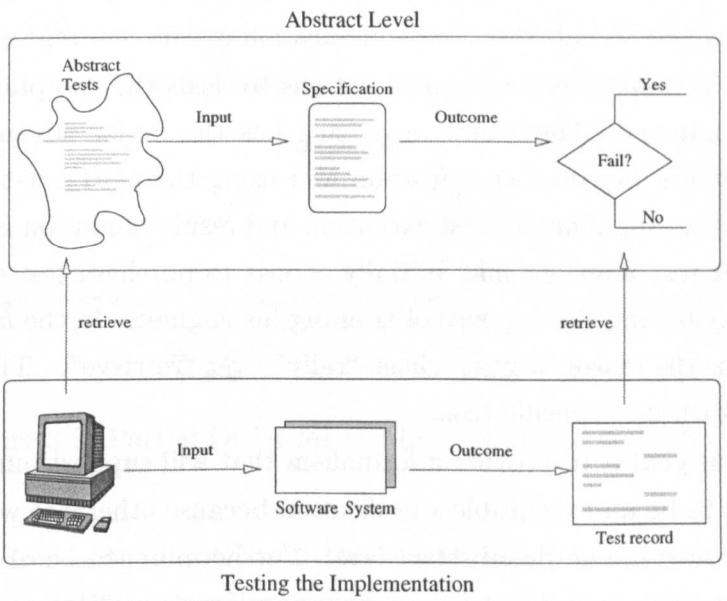


Figure 3.2: Alternative to Test Reification

which we call the "reify approach" (Figure 3.1), we apply a transformation to the expected outcome from the specification and compare it at the concrete level of the implementation with the actual outcome using a commercial test result validation tool. In the second case, which we call the "retrieve approach" (Figure 3.2 (Antoy and Hamlet, 1992)), we transform the actual outcome of the test into its abstract representation and evaluate the test at the abstract level by comparing the converted value with the abstract outcome from the specification. One goal of *hep*TEST is to be an add-on to the existing test framework. This means that the changes to the traditional test process should be minimal. A tester can continue to run the generated tests in the existing environment as usual. The tester does not have to care about the origin of the test cases. The expected outcome is produced by *hep*TEST in a format that is useful to the tester. Therefore test execution and result validation become independent of the *hep*TEST-tool. On the other hand the approach proposed in Figure 3.2 has the advantage that we can use the power of a formal system to compare actual and expected outcome. Details present in the implementation are removed during retrieval and test result validation means establishing that the equation

$$\text{actual outcome} = \text{expected outcome}$$

holds. We can use a term-rewriting system to verify that the equation holds. This approach caters for cases where one abstract outcome has many concrete representations.

Our decision is mainly lead by considerations towards the acceptance of the approach by the testing industry. The "reify" approach has the major advantage that the skills necessary to create a *hep*SPEC-specification and using the *hep*TEST-tool are only needed during the initial testing phases. Test execution and result validation is not affected by the *hep*TEST-tool. A test manager may initially choose to purchase test case generation as a "service" without having the expense of training his engineers in the *hep*TEST technology. This advantage is the reason why we chose "reify" over "retrieve". This general approach is now broken down into specific tasks.

First of all our goal is to provide a formalism that will support our aims of testing. In particular it has to be an executable specification because otherwise we would not be able to calculate the outcome at the abstract level. Furthermore, to be of any practical value, it has to support basic operations for constructing large specifications and also provide a convenient way to formalise the desired properties. We have chosen an algebraic formalism and modified it to serve our purposes creating our own specification language *hep*SPEC.

Now we will address the individual task from the testing process and discuss our ideas

for solutions. The first obstacle is the need for an order on values, as the values may be not just numbers but complex structures, over which it may be difficult to find an ordering function.

3.5.1 Test Selection Support through Partial Order on Terms

To support test selection as proposed by Beizer (1995) we need an order on values if we want to find minimal and maximal values. We could require an ordering operation for each sort from the user. This operation would become an integral part of the specification. We will not follow this approach. First, it may be difficult for the user to define a meaningful partial order for each sort. The user is distracted from analysing the system and the specification is enriched by operations which are not part of the problem description. Thus the specification becomes more complicated and difficult to maintain. Secondly, the ordering operation may be a part of the system and therefore needs to be tested itself. We cannot use the operation for test derivation as well as test execution.

In the search of a partial order inherent to the specification we consider using the subterm relation. The subterm relation defines a partial order on terms (see Section 5.2.1). For example `succ(zero)` is smaller than `succ(succ(zero))` because `succ(zero)` is a subterm of `succ(succ(zero))`. To use the subterm relation has the advantage in that we gain a partial order for each sort at no cost. Figure 3.3 illustrates the idea for the operator `add`. The drawback is that it does not always reflect our notion of an order.

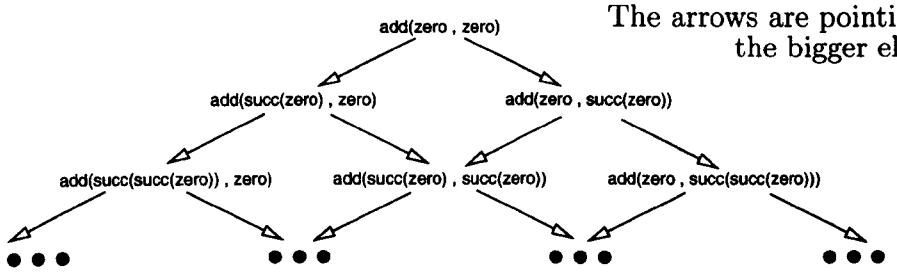


Figure 3.3: Partial Order for the Domain of the Operator `add`

The term `succ(zero)` is smaller than the term `add(succ(zero),zero)` according to the definition of subterm relation, but both terms denote the value 1 in the algebra of natural numbers. This is the result of the fact that the operator `add` does not generate new natural numbers. We defined the operator through axioms in terms of `succ` and `zero`. We can say that the operators `zero` and `succ` generate the natural numbers. We can make use

of our observation and apply this to specifications. If we distinguish between *generators* and *defined operations* then we can use the subterm relation as the partial order if we consider only terms built from generators. Terms containing only generators are called *generator terms*. This approach excludes terms built from defined operators such as **add** and reduces the number of obsolete tests. A drawback is still how axioms between generator terms should be treated. They too may cause that a value is denoted by more than one generator term. This in turn would lead to terms which will be considered as test inputs even though the denoted values are already covered or not relevant according to the testing strategy. We will discuss this topic in Chapter 4.

3.5.2 Domain Selection through Axioms

Section 2.3.4 opposes the idea of testing axioms on their own, but axioms are important in our specifications. We argue that axioms split the domains into subdomains. We re-express the Example 3.2 to emphasise our idea. Consider the specification as written in Example 3.7.

```
1     sorts Nat, Bool
2
3     oprn zero: → Nat
4          succ: Nat → Nat
5           add: Nat Nat → Nat
6          true: → Bool
7         false: → Bool
8           leq: Nat Nat → Bool
9
10    axioms a,b,c: Nat
11       if a = zero then add(a,b) = b
12       if a=succ(c) then add(a,b) = succ(add(c,b))
13       if a=zero then leq(a,b) = true
14       if c=succ(a) & b=zero then leq(c,b) = false
15       if c=succ(a) & d=succ(b) then leq(c,d) = leq(a,b)
```

Example 3.7: Algebraic Specification with Explicit Conditional Axioms

The axioms in lines 11–12 are recorded differently from the Example 3.2 to illustrate how axioms partition the domain. The domain of the operator **add** is (**Nat**, **Nat**) which is partitioned by the axioms into ({**zero**}, **Nat**) and (**Nat** \ {**zero**}, **Nat**). To partition the domain we view each premise as a case selection. This is how we lift traditional testing

strategies to the level of specifications. Let us compare this domain partitioning technique with approaches from traditional testing (Beizer, 1990).

To select tests from the domain partitions, we could use a strategy as suggested in Section 3.4. To establish minimal and maximal values, we use the partial order on terms as described above, thus we can have more than one minimal or maximal value. There is no guarantee that minimal or maximal values exist. In our example we have only minimal values. For the partition $(\{\texttt{zero}\}, \texttt{Nat})$ the minimal value is $(\texttt{zero}, \texttt{zero})$. And for the domain $(\texttt{Nat} \setminus \{\texttt{zero}\}, \texttt{Nat})$ it is $(\texttt{succ(zero)}, \texttt{zero})$. We illustrate the domain and partitions in Figure 3.4 and mark the minimal values. Because the partitions are infinite there are no maximal values.

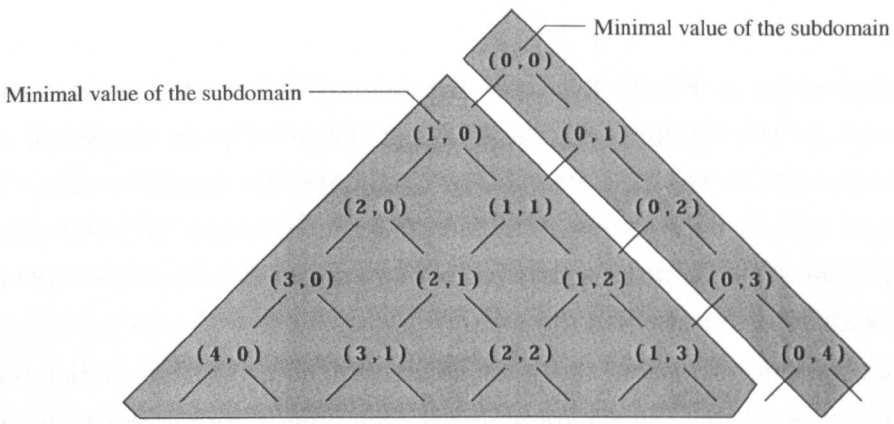


Figure 3.4: Partitions of the Domain of Operator `add`

Section 3.4 opposed the use of only minimal and maximal elements. It has been argued that this strategy exercises only the parts of the system that deal with special cases such as boundaries. Here we make a similar observation: from the two partitions we chose only two test cases, extrema. We would need to choose "average" values as well. An "average" value would be an arbitrary value from the inside of the domain. We can use the partial order on the domain again to specify that a value from inside the domain should be greater than the minima and smaller than the maxima, if they exist.

In Example 3.5 we used the auxiliary sort `Bool` and the operator `leq` to specify the domain condition for the operator `sub`. Auxiliary operators do not need to be part of the implementation, but they still contribute to the test case selection process. In our example the tests for the operator `sub` would be elements of the domain `Nat`. The domain for `sub` is restricted by the condition `leq(b,a) = true`.

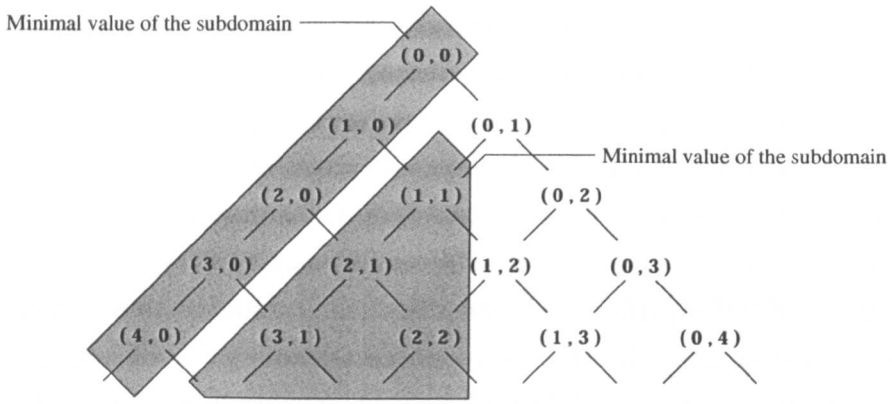


Figure 3.5: Partitions of Domain for Operator `sub`

The subdomains in Figure 3.5 show the split of the input domain. We can observe that the union of the subdomains is a subset of the set of pairs of natural numbers. The unmarked elements in the figure would be candidates for negative tests. The inputs of negative tests should be rejected. The remaining subdomains are split according to the axioms of all the other operations involved in the computation.In this simple case we would have to consider the axioms of `sub`.

The examples have shown the basic principles of using the partial order on terms to support test selection. In Chapter 6 this technique is presented in a case study and illustrates the use of partial orders in a larger example.

3.5.3 Test Reification

A criticism of the work carried out by Bougé et al. is that it does not bridge the gap between the abstract specification and the actual implementation. Woodward (1993) used this argument to introduce mutation testing to specification testing. He pointed out that the tests generated by Bougé et al. (1986) cannot be applied to the implementation and that further work is needed to do so. He argued that the tests are suitable to test the specification only and proposed in turn to use mutation testing instead.

A key contribution of this thesis is the use of homomorphisms to close the gap. In *hep*TEST two homomorphisms are distinguished, the syntactical and the semantical. The sorts and the operators from the specification are firstly related to inputs and operations from the concrete domain of the implementation. Sorts and operators may be implemented in different ways, regarding to the input and output. This is the reason behind the choice

of two homomorphisms. The homomorphisms `synI` and `synO` in Figure 3.6 represent the link from the abstract operator σ to the concrete operation f_σ where f_σ is the operation from the implementation which has been assigned to the specification operator σ.

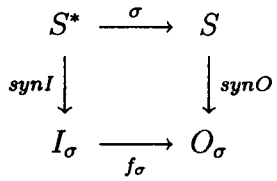


Figure 3.6: Syntactic and Semantic Homomorphisms in Testing

For a given abstract test for the operator σ the input S^* will be converted into a value from I_σ, the domain of f_σ. The test is executed and the outcome and expected outcome are compared. The abstract expected outcome is converted into a value from O_σ by the syntactic homomorphism *synO*. If the diagram commutes then the test has been successful.

The idea is to represent this homomorphism with grammars, utilising parser techniques for test case generation. An advantage is that the theoretical basis of algebraic specification is closely linked to grammars. Exploiting this means that the underlying theoretical framework is identical and therefore we can easily prove that the generated tests are logical consequences. In practice this results in reliable tests. Tests selected from a high level of abstraction are provably correctly implemented at the concrete level. This gives no rise to concerns that test reification has introduced errors.

From a practical point of view this is very important. As testing is a means of raising confidence in the correctness of the implementation, it would be a major concern if tests failed because they where incorrect. Even the opposite might be suspected: that incorrect implementations passed tests successfully because of mistakes during the reification process. This seems to be counter-productive and therefore having the possibility to prove that a test in question can be derived from the specification within a single calculus cannot be underrated.

Grammars also have the advantage of being a well-known and well-understood formalism. The ability to specify constructions in an easy, recursive way is important from an engineering point of view. In the same way we constructed the specification, values from the concrete domain are translated into abstract terms.

The Example 3.8 represents a grammar which translates numbers in binary represen-

tation into the abstract terms of the specification in Example 3.5.

```
1     exp: nat '+' nat MEANS exp = add(nat,nat1);
2        | nat '-' nat WHERE leq(nat1,nat) = true
3                      MEANS exp = sub(nat,nat1);

4     nat: digit MEANS nat = digit;
5        | nat digit MEANS nat = add(add(nat1,nat1),digit);

6   digit: '0' MEANS digit = zero;
7        | '1' MEANS digit = succ(zero);
```

Example 3.8: Utilizing Grammars to Express Homomorphism

The grammar gives the values of the algebraic specification and concrete representation. The character '0' is assigned to the value of **zero** by the production in Line 6. The semantic value of **digit** becomes **zero**. This part of the grammar can be translated into the following lines from an algebraic specification.

```
1  sort digit
2   oprn '0' → digit

3  modify specNat by specDigit according to
4   digit is Nat
5   '0' is zero
```

The sort **digit** was introduced for the nonterminal **digit**. The operator '0' is derived from the grammar rule. The nonterminals on the right hand side become the domain of the operator and the range is the nonterminal on the left hand side. In this example the domain is empty and the range is **digit**. The keyword **MEANS** initiates a modification of the original specification called **specNat** by the new specification **specDigit**. The sort **digit** is assigned to **Nat** and the operator '0' to **zero**. These transformations into the context of algebraic specifications can be used to prove the consistency of the test generation process. The special feature of our algebraic formalism, restricted domains, is catered for by the grammar using the keyword **WHERE**. After this keyword the domain restriction follows. In the context of grammars, this means that the rule can be only applied if the condition holds. For an example see Line 2 in Example 3.8.

For test reification reversed parser techniques can be applied (Purdom, 1972; Aho et al., 1985). The Figure 3.7 illustrates how the parse tree can be generated for the

test `add(succ(zero),succ(succ(zero)))`. The process is in top-down direction. The nodes are marked with the remaining values to be generated. The leafs of the parse tree constitute a test at the concrete level. The top node of the tree contains the result of the test. This result is translated using the *synO* homomorphism into an expected outcome in the correct representation.

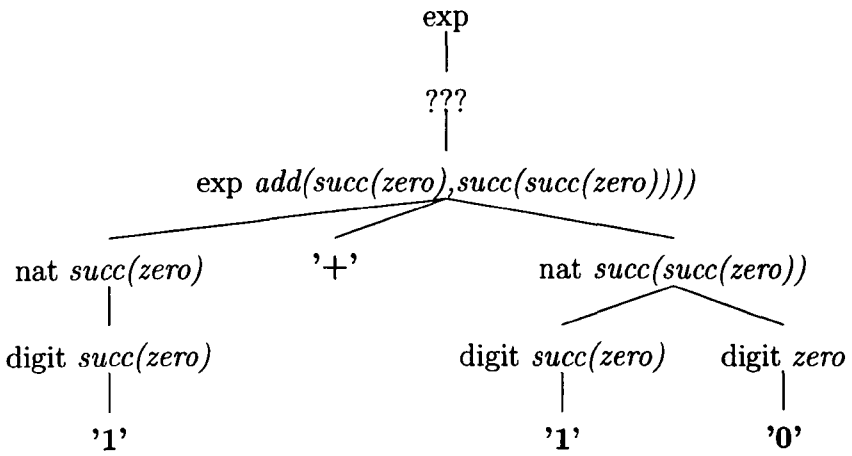


Figure 3.7: Reverse Parse Tree Generation

3.5.4 Test Execution and Validation

The particular abstract test is translated into its concrete representation by using reverse parser techniques as mentioned in Section 3.5.3. This process can be utilised to serve another purpose as well. For the test to run the system under test it has to be in a specific condition which allows the execution to take place. A variety of tasks might be needed to setup the system, such as initialisation or location of external components. But even more common, operations from the system need to be executed before a particular test can be run. If the test contains an operation with a domain restriction, then the previously executed commands would ensure, that the domain restriction of the operation holds when its associated command is executed.

It is obvious that test should therefore not only contain the inputs for a single test, but also all the necessary commands that set the system up. The same technique which is used for test reification can be used for this purpose. The generation process is not stopped after a match for the abstract test is found, but continued, generating the entire parse tree until the start symbol of the grammar is reached and all branches are terminated by

terminal symbols. The test is the sequence of terminals, now containing the setup of the test, too.

This technique can be extended to serve for test sequencing. Systems can have a long initialisation phase. To save time tests could be grouped and the generation process would produce sequences of simple tests.

Validation of tests and the attached problems have been mentioned in Section 3.3. A solution on the theoretical part was given by restricting the definitions of operations in the specification to primitive recursion. As a result of research in the area of recursion (Péter, 1967) the comparison of the tests is computable. For validation purposes the generated tests contain an expected outcome which using the *synO* homomorphism was translated into a value from the output domain of the system. Manual validation as well as automatic can compare the expected and actual outcome which are present in the same format.

All the concepts and techniques described here are formally introduced in Chapters 4 and 5 and are applied to case studies in Chapter 6.

Chapter 4

An Algebraic Specification Formalism – *hep*SPEC

In this chapter we describe the *hep*SPEC-formalism which will allow the user to specify the functions of the system under test in an implementation-independent way. We are not concerned with the full theory of algebraic specification as developed in Reichel (1987): our focus is directed towards test case generation. This is why we present only those parts of the theory necessary for a sound foundation of *hep*TEST. Proofs of theorems are generally omitted.

First we introduce some basic mathematical notation in Section 4.1 which we need throughout the remainder of the thesis. Then we present and motivate in the following sections our choice of algebraic theory focusing on the needs of testing. We introduce in Section 4.2 the well-known concept of signatures and restrict it to hep-signatures which build the foundation of *hep*SPEC. Then we need to explain the meaning of signatures which we do in Section 4.3 before we present our notion of a specification. We conclude this chapter with definitions and motivation to consider only a very specific type of model for hep-specifications. Then we have presented the algebraic theory in the form needed to build the foundations for our testing approach.

4.1 Mathematical Notation

We often need natural numbers throughout this thesis. A definition is as follows:

Definition 4.1.1 *Finite set of natural numbers.*
$N = \{0, 1, \ldots\}$ is the set of natural numbers. We say for any $n \in N$, $[n] = \{1, 2, \ldots, n\}$ if

$n > 0$ and $[0] = \varnothing$. ◊

Furthermore we have to deal with families of sorts and functions indexed by a set S.

Definition 4.1.2 *S-sorted.*
For any set S, an S-sorted set A is a family $\langle A_s \mid s \in S \rangle$ of sets indexed by S.

For any set S and any s-sorted sets A, B, an S-sorted function f is a family $\langle f_s : A_s \rightarrow B_s \rangle$ of functions indexed by S. We will write $f : A \rightarrow B$ for short.

An S-indexed system X of variables is a family of sets of variables $\langle X_s \mid s \in S \rangle$ and for each $x \in X_s$ x can take on values in A_s only. ◊

4.2 Signatures

An important concept in algebraic specifications is that of signature. The notion of types has proven useful in programming. It is reflected here in the use of sorts. To identify different objects we use different names and call them sorts. In contrast to mathematics where in most cases a single sort is sufficient, we will be dealing with many-sorted structures. We define the arity of an operator as a sequence of input sorts and an output sort, as it is insufficient in the case of many-sorted structures to talk about the number of arguments only.

Definition 4.2.1 *Signature, Sorts, Operators, and Arity.*
A *signature* $\Sigma = (S, F, arity)$ is given by:

(1) a non-empty set S the elements of which are called the *sorts* of Σ.

(2) a set F the elements of which are called the *operators* or function symbols of Σ.

(3) a mapping *arity*: $F \rightarrow S^* \times S$ which assigns

- a finite sequence $w = (s_i \mid i \in [n])$ of sorts s_i of S for every operator f of F called the *input* or *domain* of f (in Σ).
- a sort s for every operator $f \in F$, the *sort* or *range* or *output* of f.

We will write f: $w \rightarrow s$ for $arity(f) = (w, s)$. We call the sequence $s_1, \ldots, s_n, s$ the *arity* of f (in Σ) and say f is of sort s. In case $n = 0$ we will write $f :\rightarrow s$ and call f a *constant.* ◊

This definition of a signature follows the standard literature on algebraic specification(e.g. Ehrig and Mahr, 1985), but we have already emphasised in Chapter 3 that this is insufficient in our case. The problem is that many operators cannot be defined for the direct product of sets but only for a subset of it. This means we have to extend this definition of signature to the notion of hep-signature so we can describe these restrictions in a suitable manner.

Although we have not yet formally defined hep-signatures the idea has been introduced in Chapter 3.

Hep-signatures differ from signatures firstly in the way the domains of operators are defined. In the traditional case of signatures domains are direct products of sets. In Chapter 3 we argued that many operations we use do have restricted domains, namely subsets of the direct product of their input sets. Hep-signatures cater for these needs by introducing a novel way to define an operator's domain. A domain is described by a set of equations also called *domain conditions*. All values which are solutions of the domain condition form the domain of the particular operator. The idea of an operation which cannot be applied on the entire Cartesian product of carrier sets but only to a subset of this product inherently contains the temptation to call them "partial operations". We will refrain from this mainly because it induces the idea that the operations may be used on the entire Cartesian product of carrier sets. Under certain circumstances they may have the difficulty that the result is unknown, i.e. they do not yield a value. This is not the case here. An operation can be applied to a value tuple only if this tuple is an element of its domain. Compared to the classical case of total operations (e.g. Ehrig and Mahr, 1985) where the domain is implicitly constructed as the direct product of its input sets we give the possibility of explicit domain construction. This is a major difference of the theory as developed by Reichel (1987) to other algebraic theories. This is why we prefer to talk about "operators with restricted domains". An operation is total again over its restricted domain. Later when we describe the semantics in Section 4.3 we will give an example which stresses both the need for these constructs and their importance.

We need to ensure computability of domain conditions. We will introduce a restriction for the operators which can be used in a domain condition which is necessary but not sufficient.

We also need to take into account that the domain condition has to be necessary and sufficient. The necessity restricts the application of the operation in the desired way. This means that the value of the operation exists if the input belongs to its domain. In the context of operations with restricted domains and initiality (a concept to be introduced

later) this will not work as expected. Initiality causes an additional effect of minimal domains so sufficiency of the domain condition is required. Sufficiency in turn means that the input is an element of the domain if the operation has a value.

For the construction of domain conditions we need equations and henceforth terms.

Definition 4.2.2 *Σ-term.*
Let $\Sigma = (S, F)$ be a signature. We define inductively for a *variable domain* X, i.e. a family $\langle X_s \mid s \in S \rangle$ of disjunct sets of characters – the *variables* of X must be different from the operators – and for every sort $s \in S$ *Σ-terms* over S of sort s as certain character strings in the following way:

(1) For every sort s the variable $x \in X_s$ is a Σ-term over X of sort s.

(2) For every constant $f\colon \rightarrow s$ the character string f is a Σ-term over X of sort s.

(3) For every operator $f\colon (s_i \mid i \in [n]) \rightarrow s$ of Σ and Σ-terms t_i over X of sort $s_i, n > 0$ the character string

$$f(t_1, t_2, \ldots, t_n)$$

is a Σ-term over X of sort s.

(4) An object is Σ-term only if it satisfies (1), (2) or (3).

The set of all terms over Σ is denoted by $T(\Sigma, X)$, the terms of sort s by $T(\Sigma, X)_s$. Elements of $T(\Sigma, \varnothing)$ or $T(\Sigma)$ for short are called *variable-free* or *ground terms.* ◊

Definition 4.2.3 *Σ-equations.*
Let $\Sigma = (S, F)$ be a signature and X an S-sorted system of variables. We imagine for every sort $s \in S$ a symbol $\stackrel{s}{=}$ (frequently abbreviated as "=") and consider character strings of the form

$$t_1 \stackrel{s}{=} t_2,$$

which we call Σ-*equations* or *existence equations*, where t_1, t_2 are Σ-terms of sort s over X. From here on we understand a domain condition to be a set of Σ-equations. ◊

We extend the notion of signature as defined in Definition 4.2.1 to reflect the introduction of domain conditions.

Definition 4.2.4 *ep-Signature.*
An *ep-signature* $\Sigma = (S, F, arity, dom)$ is given by:

(1) a set S the elements of which are called the *sorts* of Σ.

(2) a set F the elements of which are called the *operators* or function symbols of Σ.

(3) a mapping *arity*: $F \rightarrow S^* \times S$ which assigns

- a finite sequence $w = (s_i \mid i \in [n])$ of sorts s_i of S for every operator f of F called the *input* or *domain* of f (in Σ).
- a sort s for every operator $f \in F$, the *sort* or *range* of f.

(4) a set $dom(f)$ of existence equations for every operator $f \in F$ called the domain condition of f. ◊

An operator with an empty domain condition specifies a total operation over the full Cartesian product of the functions domain sorts. The empty domain condition will not be shown in an ep-signature.

To ensure "good behaviour" (especially the usual notion of sub-algebra has to hold) of algebras defined by such an ep-signature we have to put a further restriction on the domain conditions which we discuss in Section 4.4. Furthermore, for test case generation we have to compute the domain of an operator. Consider the case where an operator f is used directly or indirectly in its own domain condition. That would lead to the problem that while computing the domain of f we suddenly face the problem that we need the domain during this particular computation. This defect can be overcome if only operators which are already defined can be used in a domain condition. Informally said, we do not permit the (direct or indirect) use of an operator in its own domain condition. This *hierarchy condition* can be syntactically stated as follows.

We define the set of operators $ops(t)$ needed in a term t:

Definition 4.2.5 *Operator Hierarchy.*

(1) If $t = x$ and $x \in X$ then

$$ops(t) = \varnothing$$

(2) If $t = f$ and f a constant then

$$ops(t) = \{f\}$$

(3) If $t = f(t_1, t_2, \ldots, t_n)$ then

$$ops(t) = \bigcup_{i \in [n]} ops(t_i) \cup \{f\}$$

(4) The set of operators needed in an equation e of form $t_1 \stackrel{s}{=} t_2$ is

$$ops(e) = ops(t_1) \cup ops(t_2).$$

(5) The set of operators needed in a set E of equations e is

$$ops(E) = \bigcup_{e \in E} ops(e).$$

◊

Definition 4.2.6 *Hierarchy condition.*
An ep-signature Σ is called *hierarchical* if the operators f of Σ can be arranged in a sequence $(f_i \mid i \in [n])$ so that the operators contained in $ops(dom(f_i))$ precede f_i in this sequence. ◊

This means that in writing down an ep-signature an operator f_1 can be used in the domain condition of an operator f_2 if, and only if, f_1 appears textually before f_2. This is the same type of restriction as usually applied in programming languages: before one can use an item it has to be declared.

A further restriction on ep-signatures which we will consider is sensibility. Formally:

Definition 4.2.7 *Sensible.*
A ep-signature is called *sensible* if it admits at least one ground term for each sort. ◊

This is a commonly used condition to ensure sensible models (Wirsing, 1990). From now on we will consider only sensible ep-signatures, others being of no interest to us. A sort which does not admit at least one ground term would require user interference during test case generation giving the user the task to provide a value for this sort. This is against our goal of high level automation. There are three necessary conditions to ensure sensibility:

(1) For each sort $s \in S$ there must be at least one operator which has s as its sort.

(2) For every argument (if any) of this operator at least one ground term must exist.

(3) For operators with restricted conditions the domain conditions must at least have one solution consisting of ground terms only.

The first two conditions can be checked syntactically.

An ep-signature $\Sigma = (S, F, arity, dom)$ satisfying these restrictions is henceforth called a *hep-signature* where hep stands for "hierarchical equationally-partial". The name is used for historical reasons and has been proposed by Reichel (1987). From now on we will consider only hep-signatures.

Usually in algebraic specification a signature $\Sigma = (S, F, arity)$ is assigned a meaning by associating a set A_s with every sort s of S (the carrier set of s) and a mapping ϕ with every operator f of F compatible to the arity of f. In the case of hep-signatures this requires more care due to the domain conditions. In Section 4.3 we present the formal definition of a model of a hep-signature. But in general we are not interested in any given operation but in operations with certain properties. These properties are stated by axioms interrelating operators. We will not use equations as axioms as in other styles of algebraic specification but elementary implications (simplified conditional equations) instead. The introduction of operations with restricted domains implies that an operator may occur in a term as defined above if, and only if, its applicability is ensured. By means of conditional equations we care about the "definedness" of both sides of an equation.

Definition 4.2.8 *Σ-elementary implications.*
Let $\Sigma = (S, F, arity, dom)$ be a hep-signature and X an S-sorted system of variables. $G \rightarrow t_1 \stackrel{s}{=} t_2$ is called *Σ-elementary implication* over X with the finite set G of existence equations as *premises* and with $t_1 \stackrel{s}{=} t_2$ as its *conclusion.* $\Diamond$

A specification SPC=(Σ, Ax) is a hep-signature $\Sigma = (S, F, arity, dom)$ together with a set Ax of elementary implications over Σ serving as axioms. The specification SPC is not called a hep-specification because we want to put further restrictions on the axioms before we use that name. If we permit any axioms then it might be that these axioms cause - unintentionally - additional domain conditions. This would render the process of test data generation as described in the next chapter considerably more difficult. But to give such guidelines in a formal manner we need to define what it means to say that an axiom holds.

4.3 Semantics of Specifications

In this section we will assign a meaning to signatures. The meaning is traditionally explained in terms of those algebras satisfying the axioms.

Our axioms are mainly conditional equations which are built from equations and these in turn are constructed from terms. To satisfy an equation the terms on the right-hand side and on the left-hand side have to denote the same value. So we have to define exactly what the value of a term is. At first we have to extend the concept of domain condition from operators to terms as well. We do this inductively.

Definition 4.3.1 *Domain conditions of terms and equations.*
Let $\Sigma = (S, F, arity, dom)$ be a hep-signature and X an S-sorted system of variables.

(1) If $t = x$ and $x \in X$ then

$$dom(t) = \varnothing$$

(2) If $t = f$ and f a constant then

$$dom(t) = dom(f)$$

(3) If $t = f(t_1, t_2, \ldots, t_n)$ then

$$dom(t) = \bigcup_{i \in [n]} dom(t_i) \cup dom(f)$$

(4) The domain condition of an equation $e = t_1 \stackrel{s}{=} t_2$ is

$$dom(e) = dom(t_1) \cup dom(t_2).$$

(5) The domain condition of an equation set E is

$$dom(E) = \bigcup_{e \in E} dom(e).$$

◊

Definition 4.3.2 *Σ-algebra.*
Let $\Sigma = (S, F)$ be a signature. A Σ-algebra $\underline{\mathbf{A}}$ is given by a family $\langle A_s \mid s \in S \rangle$ of *carrier*

sets A_s and a family $\langle f^A \mid f \in F\rangle$ of *operations* f^A: $def(f^A) \to A_s$ with $def(f^A) \subseteq A_w = (A_{s_i} \times ... \times A_{s_n} \mid i \in [n])$ for f: $w \to s$ (in Σ). ◊

To define what the meaning (i.e. value) of a term is we need the notion of assignment associating with every variable x a value from some set.

Definition 4.3.3 *Assignment.*
Let $\Sigma = (S, F)$ be a signature. We consider an *assignment* $\alpha = \langle \alpha_s : X_s \to A_s \mid s \in S\rangle$ of X in $\underline{\mathbf{A}}$ for a Σ-algebra $\underline{\mathbf{A}}$ and a variable domain X to be constructed by assigning every variable x in X of sort s an element $\alpha_s(x) \in A_s$ of the corresponding carrier set. ◊

By means of an assignment we can compute the value of a term.

Definition 4.3.4 *Value of a term.*
Let $\Sigma = (S, F)$ be a signature, X an S-sorted system of variables, $\underline{\mathbf{A}}$ a Σ-algebra and α an assignment.

(1) If $t = x$ and $x \in X$ then

$$val(t_A) = \alpha(x)$$

(2) If $t = f$ and f a constant then

$$val(t_A) = f^A$$

(3) If $t = f(t_1, t_2, \ldots, t_n)$, each t_i has a value, and the tuple of t_i values is an element of the domain of f^A then

$$val(t_A) = f^A(val(t_1), val(t_2), \ldots, val(t_n))$$ ◊

An assignment is considered a solution of an equation, if both sides of the equation are defined (have a value) and these values are equal.

Definition 4.3.5 *Solution of an equation.*
An assignment α of X in $\underline{\mathbf{A}}$ is called a *solution* in $\underline{\mathbf{A}}$ of the Σ-equation $t_1 \stackrel{s}{=} t_2$ if the value of t_1 under α exists, if the value of t_2 under α exists and if both values are equal. ◊

We write also $(A, \alpha) \models (t_1 \stackrel{s}{=} t_2)$ to denote the fact that α is a solution of $t_1 \stackrel{s}{=} t_2$ in $\underline{\mathbf{A}}$.

An assignment α is considered to be a solution of a set of existence equations E if and only if α is solution for each $e \in E$. The set of all solutions of E will be briefly denoted by A_E.

In the literature this is called *weak equality*. In contrast *strong equality* also holds when both sides are undefined. We use weak equality which allows us to require the definedness of t by simply writing down $t = t$ (Wirsing, 1990).

We introduced equations and elementary implications as a means of expressing the desired properties of the system under test without mentioning how this is achieved. Now we have developed the theory far enough to state what it means to say that an axiom holds.

Definition 4.3.6 *Satisfying an axiom.*
A Σ-axiom (conditional equation) $G \rightarrow e$ over a variable domain X *holds* or is *satisfied* in a Σ-algebra $\underline{\mathbf{A}}$ if, and only if, every assignment of X in $\underline{\mathbf{A}}$ being a solution of all premises in G of $G \rightarrow e$ is also a solution of the conclusion of $G \rightarrow e$. ◊

Using the above introduced notion of solution set we can reformulate this definition as $A_G \subseteq A_{t_1 \stackrel{s}{=} t_2}$. The fact that an axiom $G \rightarrow t_1 \stackrel{s}{=} t_2$ holds in $\underline{\mathbf{A}}$ is denoted by $A \models (G \rightarrow t_1 \stackrel{s}{=} t_2)$, too.

Definition 4.3.7 *Θ-algebra.*
Let $\Theta = (\Sigma, Ax)$ be a hep-signature together with a finite set Ax of axioms. Let $\underline{\mathbf{A}}$ be a Σ-algebra. $\underline{\mathbf{A}}$ is called a Θ – *algebra* if, and only if,

(1) $A_{\mathrm{dom}(f^A)} = A_{\mathrm{def}(f)}$ for every $f \in F$.

(2) all axioms of Ax hold in $\underline{\mathbf{A}}$. ◊

In other words, a Σ – *algebra* $\underline{\mathbf{A}}$ is a Θ – *algebra* if the domains of all operations of $\underline{\mathbf{A}}$ coincide with the solution set of the domain conditions of its corresponding operators and if (of course) the axioms hold. $\underline{\mathbf{A}}$ is called an interpretation or model of Θ. We express this fact by $\underline{\mathbf{A}} \models Ax$ ($\underline{\mathbf{A}}$ satisfies Ax). An elementary implication I is said to be a logical consequence of the axioms if whenever $\underline{\mathbf{A}}$ satisfies the axioms it satisfies I as well. We write $Ax \models I$.

An important notion in algebra is the concept of homomorphism. It provides for the characterisation of "similarly working" algebras in the sense that the operations act in the same manner. With h a family of mappings $\langle h_s : A_s \rightarrow B_s \mid s \in S \rangle$ between Θ–*algebras* $\underline{\mathbf{A}}$

and $\underline{\mathbf{B}}$ this key property is usually expressed as $h_s(f_A(a_1, ..., a_n)) = f_B(h_{s_1}(a_1), ..., h_{s_n}(a_n))$. Remember that either side of this equation can be undefined. So we have to paraphrase this in the following definition.

Definition 4.3.8 *Σ-homomorphism.*
Let $\Sigma = (S, F, arity, dom)$ be a hep-signature. A *homomorphism* h: $\underline{\mathbf{A}} \to \underline{\mathbf{B}}$ for Σ-algebras $\underline{\mathbf{A}}$, $\underline{\mathbf{B}}$ is a family $\langle h_s \mid s \in S \rangle$ of maps h_s: $A_s \to B_s$ with the property that for every operator f: $(s_i \mid i \in [n]) \to s$ of Σ and every tuple of values $\mathbf{a} = \langle a_1, ..., a_n \rangle$ the following is valid:

$$f_A(a_1, ..., a_n) = a \implies f_B(h_{s_1}(a_1), ..., h_{s_n}(a_n)) = h_s(a).$$

This includes:

$$\langle a_1, ..., a_n \rangle \in \mathrm{def}\,(f^A) \implies \langle h_{s_1}(a_1), ..., h_{s_n}(a_n) \rangle \in \mathrm{def}\,(f_B). \qquad \Diamond$$

If additionally

$$\langle h_{s_1}(a_1), ..., h_{s_n}(a_n) \rangle \in \mathrm{def}\,(f_B) \implies \langle a_1, ..., a_n \rangle \in \mathrm{def}\,(f^A)$$

holds then the homomorphism is called closed. A bijective homomorphism, that is a homomorphism where each h_s is bijective, is called *isomorphism.*

With the help of homomorphisms we are able to further narrow the class of Θ-algebras to which we are interested in. Our aim is the automatic generation of test data, i.e. certain representations of values, for example character strings denoting numbers. Therefore we need a denotation for every value because we cannot rely on user interference if such a denotation is not given. Initial algebras provide the vehicle towards this end.

Definition 4.3.9 *Initial models.*
A Θ-model $\underline{\mathbf{A}}$ is called an *initial* Θ-model if there is exactly one Σ-homomorphism h: $\underline{\mathbf{A}} \to \underline{\mathbf{B}}$ to any Θ-model $\underline{\mathbf{B}}$. $\Diamond$

Initial algebras are very interesting to us because they are unique up to isomorphism. We might say that the behaviour is fixed and the "variable part" is the representation only. Imagine the algebra $\underline{\mathbf{Nat}}$ of natural numbers with 0, *succ* and + as operations. The initial models are different only in the representation of the natural numbers (say as binary, decimal or Roman digits) but not in the behaviour of the operations. They have two

important properties often paraphrased with "no junk - no confusion" (Ehrig and Mahr, 1985; Reichel, 1987). Formally we state for initial algebras:

- Every value is represented by at least one ground-term.
- Different ground terms denote equal values if and only if this equality is a logical consequence of the axioms.

4.4 Hep-specifications

We can now return to the discussion at the end of the Section 4.2 on hep-signatures where we told about further restrictions on axioms. We start with restricting the form of axioms. Up to now any term may be used in an equation. But sometimes it can be very difficult to understand what an equation means resp. what are the effects of introducing a new equation on the effects already given equations have. This becomes especially important if we want to construct specifications in a step-wise manner as we will do later on. Therefore it seems to be reasonable to use only axioms constructed according to the scheme of primitive recursion. One might argue that this restriction is too strong. But as Guttag and Horning (1978) have mentioned: "In programming, the occasions when one has a need of a non-primitive recursive function seem to be very rare indeed". This argument applies even more in our context of test data generation because we need a constructive approach. It is not sufficient here to require the existence of an element. We must give an instruction how to construct this element explicitly.

Definition 4.4.1 *Primitive-recursive axioms.*
An axiom Ax is called $primitiv - recursive$ if the equations of the premise and conclusion of Ax are defined inductively as follows:

(1) $h(0, \mathbf{x}) = f(\mathbf{x})$

(2) $h(succ(y), \mathbf{x}) = g(\mathbf{x}, y, h(y, \mathbf{x}))$

where 0 stands for any constant function, *succ* is a successor function in the sense of structural induction, f and g denote already defined functions. ◊

Let us now try to order the axioms in the following way. An axiom a_1 is "smaller" than an axiom a_2 if the domain condition of the conclusion of a_2 is a consequence of a_1 (and possibly other axioms smaller than a_2). Formally we define a *stable* set of axioms.

Definition 4.4.2 *Stable set of axioms.*
Let $\Theta = (\Sigma, Ax)$ be a hep-signature together with a set Ax of axioms. Ax is called *stable* if, and only if, Ax can be partially ordered so that for all axioms a: $G \rightarrow t_1 \stackrel{s}{=} t_2 \in Ax$ the following holds: $G \rightarrow \mathrm{dom}\,(t_1 \stackrel{s}{=} t_2)$ is a consequence of all those axioms smaller than a. $\Diamond$

In a slightly loose manner this means: if the premises of an axiom hold then the domain condition of the conclusion is satisfied, too. There is a further interesting observation on "practical" algebras, i.e. the opposite may be equally true: If the domain condition of the conclusion is satisfied then the premises of an axiom hold likewise. This amounts to a semantic equivalence of premises and domain conditions. We insist in using stable sets of axioms satisfying this latter condition and define therefore hep-axioms.

Definition 4.4.3 *Set of hep-axioms.*
Let $\Theta = (\Sigma, Ax)$ be a hep-signature together with a stable set Ax of axioms. Ax is called a set of *hep-axioms* if for all axioms a: $G \rightarrow t_1 \stackrel{s}{=} t_2 \in Ax$ the following holds: $\mathrm{dom}\,(t_1 \stackrel{s}{=} t_2) \rightarrow G$ is a consequence of all those axioms smaller than a. $\Diamond$

Definition 4.4.4 *hep-Specifications.*
Let $\Theta = (\Sigma, Ax)$ be a hep-signature together with a set Ax of hep-axioms. We call Θ a hep-specification. $\Diamond$

Up to now we considered specifications as monolithic blocks. In practice specifications with a large number of sorts and operators become unmanageable. In the same way as programs are split into modules, we want to split specifications into parts. These parts become building blocks for larger specifications. Thus we need to provide operations for constructing specifications out-of other specifications. The need for such operations over specifications has been stressed in the literature and led to the principle of "stepwise construction" (Wirth, 1971). But before discussing this in detail let's divide the set of operators of Σ into generators and manipulators. Take again Nat as an example. We observe that all values of Nat are constructed by the operations 0 and *succ* alone. They serve as generators whereas + might be called a manipulator. This distinction is mainly for a better comprehensibility but in our context of test data generation it can ease life considerably. When we talk about domains in the future then it becomes sufficient to consider generator terms.

We generalise this observation in the following definition.

Definition 4.4.5 *Generators and Manipulators.*
For a signature $\Sigma = (S, F)$ we will distinguish operators from F as *generators* $g \in G$ and

manipulators $m \in M$ with $G \cap M = \varnothing$ and $F = G \cup M$. G is a set of generators iff the value of every term can be expressed by a term containing only symbols in G and there does not exists a proper subset of G which satisfies this property, too. Terms containing only operators from G are called ground generator terms. ◊

We write $G(\Theta)$ for the set of generators of Θ, $M(\Theta)$ for the set of manipulators and $Ax(\Theta)$ for the set of axioms. It is important to mention that this distinction reflects the intention of the designer rather than providing a checkable property. This suggests the idea to define in a first step all the things we want to work with, i.e. to specify all the sorts and their generators. In a second step we enhance this specification by defining the operations realizing the required behaviour, i.e. the manipulators without defining new sorts. But on behalf of initial satisfaction we must take care that no existing objects are identified, no new objects are created and the domains of existing operations are not changed. To state these conditions in a precise way we have to define them formally. We start with the notion of *specification enhancement.*

Definition 4.4.6 *Specification enhancement.*
Let $\Theta = (\Sigma, Ax)$ and $\Theta' = (\Sigma', Ax')$ be two hep-specifications. Θ' is an *enhancement* of Θ if $S' \supseteq S$, $F' \supseteq F$ and $Ax' \supseteq Ax$. We write $\Theta' \sqsupseteq \Theta$. ◊

Let $\underline{\mathbf{A}}$ be a Θ'-algebra. We can define the Θ-part of $\underline{\mathbf{A}}$ called Θ-reduct.

Definition 4.4.7 Θ*-reduct.*
Let Θ' be an enhancement of Θ. Then we get the Θ-reduct $\underline{\mathbf{A}}\downarrow \Theta$ of a Θ'-algebra $\underline{\mathbf{A}}$ by

(1) $(\underline{\mathbf{A}}\downarrow \Theta)_s = A_s$ for $s \in S(\Theta)$.

(2) $f_{A\downarrow\Theta} = f^A$ for every operator $f \in F(\Theta)$. ◊

Informally we consider only those carrier sets and operations corresponding to sorts and operators of Θ and we "forget" all sets and operations which are in Θ' but not in Θ. The remaining sets and operations must not have changed. In a similar way we can restrict a Θ'-homomorphism h' to its corresponding Θ-homomorphism h by setting $h = \{(h' \downarrow \Theta)_s \mid s \in S(\Theta)\}$.

We want to extend the concept of initiality from a single specification to specification enhancements to keep all the properties useful for the automatic generation of test data and tests. This is achieved with the principle of *free extension.*

Definition 4.4.8 *Free extension.*
Let Θ, Θ' be hep-specifications and $\Theta' \sqsupseteq \Theta$. The Θ'-algebra $\underline{B}$ is a *free extension* of the Θ-algebra $\underline{A}$ if, and only if,

- $\underline{B}\downarrow \Theta = \underline{A}$
- for any Θ'-algebra $\underline{D}$ and any Θ-homomorphism g: $A \to D \downarrow \Theta$ there is exactly one Θ'-homomorphism h: $B \to D$ such that the following diagram commutes

$$\begin{array}{ccc} \underline{B} & \xrightarrow{h:\ B \to D} & \underline{D} \\ \downarrow & & \downarrow \\ \underline{A} = \underline{B}\downarrow\Theta & \xrightarrow[g:\ A \to D\downarrow\Theta]{} & \underline{D}\downarrow\Theta \end{array}$$

◊

The requirement of initiality could be expressed in this context as $\Theta' \sqsupseteq \varnothing$, the empty specification.

In Section 3.3 we argued against using error values to model partiality. Then we promised to show formally the difficulties arising from using error values. By now we have presented the definitions necessary to prove our claim. Let us make an excursion.

```
Graph is definition
 sorts Node, Arc
 oprn begin(Arc) → Node
      end(Arc) → Node
end Graph
```

Example 4.1: `Graph` Specification

Consider the specification in Example 4.4. `Graph` is the base specification. It defines the sorts `Arc` and `Node` and two operations being `begin` and `end`.

We can extend `Graph` to `Path1` as shown in Example 4.2. `Path1` is a free extension of `Graph`. `Path1` uses partial functions to monitor the fact, that an arc can only be appended to a path if the begin of the arc is the end of the path.

Let us consider now another extension of `Graph`. Example 4.3 provides `Path2`. `Path2` uses error values to monitor the same problem as in `Path1`.

Then consider the following two Algebras A and B (see Figure 4.1). Algebra A consists of the nodes *1, 2, 3* and *4* and the arcs a and b. Algebra B consists of the nodes *I*, *II* and *III* and the arcs α and β.

```
Path1 is Graph with definition
 sorts Path
 oprn source(Path) → Node
      target(Path) → Node
      nil(Node) → Path
      append(p:Path, a:Arc
       iff target(p) = begin(a)) → Path
axioms a: Node, x: Arc, p,q: Path
 source(nil(a)) = target(nil(a)) = a
 if append(p,x) = q
  then source(q) = source(p)
 if append(p,x) = q
  then target(q) = end(x)
end Path1
```

Example 4.2: Path1 Specification

```
Path2 is Graph with definition
 sorts Path
 oprn source(Path) → Node
      target(Path) → Node
      nil(Node) → Path
      append(Path,Arc) → Path
      errorPath(Node,Node) → Path
 axioms a,b: Node, x: Arc, p,q: Path
  source(nil(a)) = target(nil(a)) = a
  if q = append(p,x)
   then source(q) = source(p)
  if q = append(p,x)
   then target(q) = end(x)
  source(errorPath(a,b)) = a
  target(errorPath(a,b)) = b
  append(errorPath(a,b),x) = errorPath(a,end(x)
  if target(p) ≡ begin(a) = false
   then append(p,a) = errorPath(source(p),end(a))
end Path2
```

Example 4.3: Path2 Specification

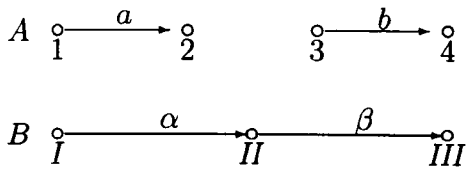


Figure 4.1: Path2-Algebras $\underline{\mathbf{A}}$ and $\underline{\mathbf{B}}$

We could define $h : A \downarrow Graph \rightarrow B \downarrow Graph$ as follows:

$$h(1) = I \quad h(2) = h(3) = II \quad h(4) = III$$
$$h(a) = a \qquad\qquad h(b) = b$$

Theorem 4.4.1
Path2 is not a free extension of Graph w.r.t. $A \downarrow Graph$. ◊

Proof
Let B and h: $A \downarrow Graph \rightarrow B \downarrow Graph$ be defined as above. Then we must show that there does not exist a unique extension of h to a homomorphism h': $A \rightarrow B$.

Let h' be an arbitrary mapping extending the homomorphism h. For a contradiction assume that h' is a homomorphism. Then

$$h'(append_A(append_A(nil_A(1), a), b)) = append_B(append_B(nil_B(h'(1)), h'(a)), h'(b))$$

by assumption that h' is a homomorphism,

$$= append_B(append_B(nil_B(I), \alpha), \beta)$$

by definition of h and the fact that h' extends h,

$$\neq errorPath_B(I, III)$$

by definition of B

$$= h'(errorPath_A(1, 4))$$

assuming h' is a homomorphism,

$$= h'(append_A(append_A(nil_A(1), a), b))$$

by definition of A. Thus the initial assumption that h' is a homomorphism produces a contradiction.

We have shown that h can not be extended uniquely and by that `Path2` is not a free extension of `Graph`. □

Thus our choice considering the important role partiality is playing in this approach is justified. Nevertheless not every specification enhancement has a free extension. We consider only *consistent enhancements* which have a free extension.

Definition 4.4.9 *Consistent enhancement.*
Let Θ, Θ' be hep-specifications and $\Theta' \sqsupseteq \Theta$. Θ' is called a *consistent enhancement* of Θ if, and only if, for every Θ'-model $\underline{\mathbf{B}}$ the Θ-reduct $\underline{\mathbf{B}}{\downarrow}\,\Theta$ is a Θ-model. If Θ' is a consistent enhancement of Θ, then every Θ'-model $\underline{\mathbf{B}}$ is a free extension of the Θ-reduct of $\underline{\mathbf{B}}$. ◊

Not every enhancement is consistent. Consider for example the specification `Nat` with sorts `nat`, `bool` and the operators `0:` → `nat`, `succ:` `nat` → `nat`, `true:` → `bool` and `false:` → `bool`. The initial model $\underline{\mathbf{A}}$ of `Nat` has the natural numbers $\{0, 1, \ldots\}$ as elements for the sort `nat` and two values say T and F for the sort `bool`. We enhance `Nat` to the specification `NatEq` by a new function `eq:` `nat, nat` → `bool` and the axiom `eq(x,x) = true`. In accordance with initiality a `NatEq`-model contains new values of sort `bool` for all pairs of different natural numbers. So $eq^A(0,0)$ would yield T, but $eq^A(0,1)$ denotes a new value of sort `bool`. The carrier of `bool` contains besides T and F infinitely many new values. Building the `Nat`-reduct of the `NatEq`-model shows that the `bool`-carrier has been changed. But that means that the `Nat`-reduct of the `NatEq`-model is not a `Nat`-model.

Let us now return to the problem of constructing specifications step-by-step. We start with the operation join, taking two specifications and combining them into one.

Definition 4.4.10 *Join of hep-specifications.*
Let Θ, Θ' be hep-specifications and $\Theta' \sqsupseteq \Theta$. We define the *join* of Θ and Θ' as $\Theta \uplus \Theta' = ((\Sigma \cup \Sigma'), (Ax \cup Ax'))$. ◊

Note: Not every join yields a hep-specification. If the sets of sorts, the sets of operators and the sets of axioms are disjoint then $\Theta \uplus \Theta'$ is a hep-specification. But consider two

specifications both of which make use of a specification BOOL. If these specifications are joined then the sort bool exists twice and might even have different axioms. It is impossible to say if two sets of axioms describe the same property in general. This gives now rise to the problem which axioms, which operators and which sorts to use for the joined specification. This problem has two solutions. First there is a possibility that although the same name has been chosen the meaning is different: the solution is renaming to resolve the ambiguity. In the case that a specification has been used at different places to denote the same thing it is sufficient to restrict the use of such specifications. We require that each sort and each operator are defined only once. This is a condition which can be checked syntactically again.

A more interesting construction is the enhancement of a given specification by new operators, i.e. a functional enrichment by introducing new manipulators. Suppose that we have done the first step and defined some sorts together with their generators. We can now introduce processing operations (manipulators in our terminology) with an *enhancement by definition*. It is important to note that enhancement by definition does not allow for new sorts. A number of axioms which assign the outcome of the new operations a meaning in terms of values denoted by generators is required.

Definition 4.4.11 *Enhancement by definition.*
Let Θ, Θ' be hep-specifications and $\Theta' \sqsupseteq \Theta$ a consistent enhancement. Θ' is an *enhancement by definition* of Θ if $G(\Theta') = G(\Theta)$, $M(\Theta') \supset M(\Theta)$ and $Ax(\Theta') \supset Ax(\Theta)$. $\Diamond$

Remember that nothing already given may be changed. Unfortunately there is no algorithmic procedure to ensure that this requirement is met. In our opinion (considering the current state of theory) the only way is to use equations as simple as possible so one can understand what an equation means. This was the main reason to restrict the form of axioms to the primitive-recursive scheme.

In contrast an *enhancement by generation* introduces new sorts and henceforth generators. We require at least one new sort and its generators for an enhancement by generation. This ensures that the resulting specification is sensible if the original has this property.

Definition 4.4.12 *Enhancement by generation.*
Let Θ, Θ' be hep-specifications and $\Theta' \sqsupseteq \Theta$ a consistent enhancement. Θ' is an *enhancement by generation* of Θ if $G(\Theta') \supset G(\Theta)$, $M(\Theta') = M(\Theta)$ and $Ax(\Theta') \supseteq Ax(\Theta)$ and no generator $g \in G(\Theta') - G(\Theta)$ generates values of any sort $s \in S(\Theta)$. $\Diamond$

We may view enhancements by generation as the construction of building blocks and enhancements by definition as manipulators of items defined in those building blocks.

4.5 Ground-term Algebras

As already mentioned in Chapter 3 our aim is the automation of the test process. A specification of the function to be tested must provide for the generation of test data and the computation of the expected outcome. The only material we can work with to reach this end are terms of some kind because we must "write down" our test cases. The concrete form of these terms is not of interest here so we can remain fairly vague. A suitable candidate for our purpose are therefore the abstract terms we have constructed in this chapter. Considering the goal of automation we must restrict ourselves to the use of ground or variable-free terms. The presence of variables in terms would require user intervention to get a value for these variables. This is clearly something we have to avoid.

We have to define the operations of Θ over ground terms as $f^{T(\sigma)} : T_1 \times \ldots \times T_n \rightarrow T_s$ with T_i sets of ground terms of appropriate sort s.

Definition 4.5.1 *Θ-ground term algebra.*
Let $\Theta = (S, F, arity, dom)$ be a hep-signature.

(1) If f is a constant then

$$f^{T(\Sigma)} = f.$$

(2) If f has the arity $s_1, s_2, \ldots, s_n \rightarrow s$, $n > 0$, every t_i is a ground term of sort s_i and the tuple of t_i values is an element of the domain of $f^{T(\Sigma)}$ then

$$f^{T(\Sigma)}(t_1, ..., t_n) = f(t_1, ..., t_n). \qquad \Diamond$$

Remember that we do not need an assignment to compute the value of a term thanks to the absence of variables.

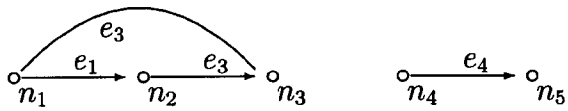
The ground term algebra $T(\Sigma)$ for a hep-signature Σ is in general not initial. Consider as example again the algebra **Nat** of natural numbers with 0, *succ* and + as operations. The operations are defined by the axioms $n+0 = n$, $n+succ(m) = succ(n+m)$. The axioms identify for example the terms $succ(0)$ and $succ(0)+0$. But the term algebra contains both

terms succ(0) and succ(0)+0 which contradicts initiality: The ground term algebra is not isomorphic to the algebra of natural numbers.

The problem is somewhat different in our context. We subdivided the operators into generators and manipulators so we have to consider only ground terms consisting exclusively of generators. If axioms interrelating these generators are present then we face the above mentioned problems. The usual way in algebra to solve this problem is the transition to quotient algebras by factoring out all terms having the same value on behalf of the axioms in one equivalence class. Unfortunately this relatively simple approach does not work in the case of partial algebras as is shown in Reichel (1987, p. 92).

```
Graph is
  sorts Node, Edge
  oprn
     begin(Edge) → Node
     end(Edge) → Node
     combine(Edge x, Edge y) → Edge
              iff end(x) = begin(y)
end Graph
```

Example 4.4: *hep*SPEC-specification of Graph



$$Node^A = \{n_1, n_2, n_3, n_4, n_5\}\ Edge^A = \{e_1, e_2, e_3, e_4\}$$

$$begin^A(e_1) = n_1,\ begin^A(e_2) = n_2,\ begin^A(e_3) = n_1,\ begin^A(e_4) = n_4$$

$$end^A(e_1) = n_2,\ end^A(e_2) = n_3,\ end^A(e_3) = n_3,\ end^A(e_4) = n_5$$

Figure 4.2: Graph-Algebra $\underline{A}$ of *hep*SPEC-specification Graph

Let us take a closer look at an example to explain the difficulties. We start with the specification in Example 4.4. We have sorts Node and Edge, the operation begin

associating a node with an edge, the operation **end** associating a node with an edge, too, and the operation **combine** computing an edge out of an edge x and an edge y iff the end of edge x ist the begin of edge y. A model of **Graph** is the **Graph**-algebra $\underline{\mathbf{A}}$ given in Figure 4.2. According to the domain condition the function *combine* is defined as $combine^A(e_1, e_2) = e_3$. (To avoid confusion: We do not append edge y to edge x making a path from $begin(x)$ to $end(y)$ but rather compute a "shortcut" between these two nodes.)

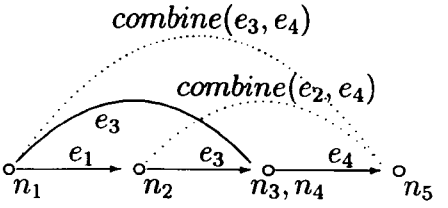


Figure 4.3: Representation of Quotient-Algebra $\underline{\mathbf{A}}|_\varrho$

Consider an S-equivalence relation $\varrho = (\varrho_{Node}, \varrho_{Edge})$ with $\varrho_{Node}\{(n_3, n_4)\}$ and $\varrho_{Edge} = \varnothing$ induced by an axiom n3 = n4. But now we could combine more edges. The pairs $(e2, e4)$ and $(e3, e4)$ are additional solutions of the domain condition $end(x) = begin(y)$ of *combine* (see Figure 4.3). The problem is there are no equivalence classes in $\underline{\mathbf{A}}|_\varrho$ denoting the results of combining (e_2, e_4) and (e_3, e_4). This example demonstrates that the usual elementwise construction of quotient algebras is generally impossible if operations with restricted domains are present.

Reichel (1987, p. 120) suggests the use of natural homomorphisms to construct quotient-algebras. But we will not pursue the matter here because another argument in favouring axiom-less generators is the use of the system LoFT to generate our test cases. As stated in Bernot et al. (1991b, p. 27) axioms defining a generator are not allowed: The top symbol of the left-hand side of an equation (or conclusion of a conditional axiom) must be a defined symbol or manipulator in our terminology.

These difficulties lead us to our decision to forbid axioms identifying generator terms: We choose the special algebra of ground generator terms or ground term-algebra for short as the only allowed models for our specifications.

Chapter 5

Testing Theory — *hep*TEST

In practice testing is achieved by executing a system with test inputs and observing the outcome. The result of such a test is judged as being successful if the expected outcome was observed otherwise it is said that the test failed. Figure 5.1 illustrates this approach.

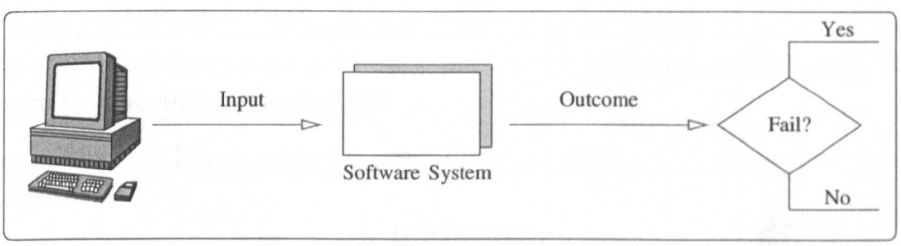


Figure 5.1: Traditional Testing Approach

Using the formalism we developed in Chapter 4 we can produce a similar diagram. The idea of *hep*TEST is to combine these two diagrams and to use the testing philosophy developed for the traditional approach in an automated setting. The result of the combination is presented in Figure 5.2.

We can split the *hep*TEST approach into the following individual tasks:

- Creating the specification
- Test input generation
- Test set selection
- Expected outcome generation

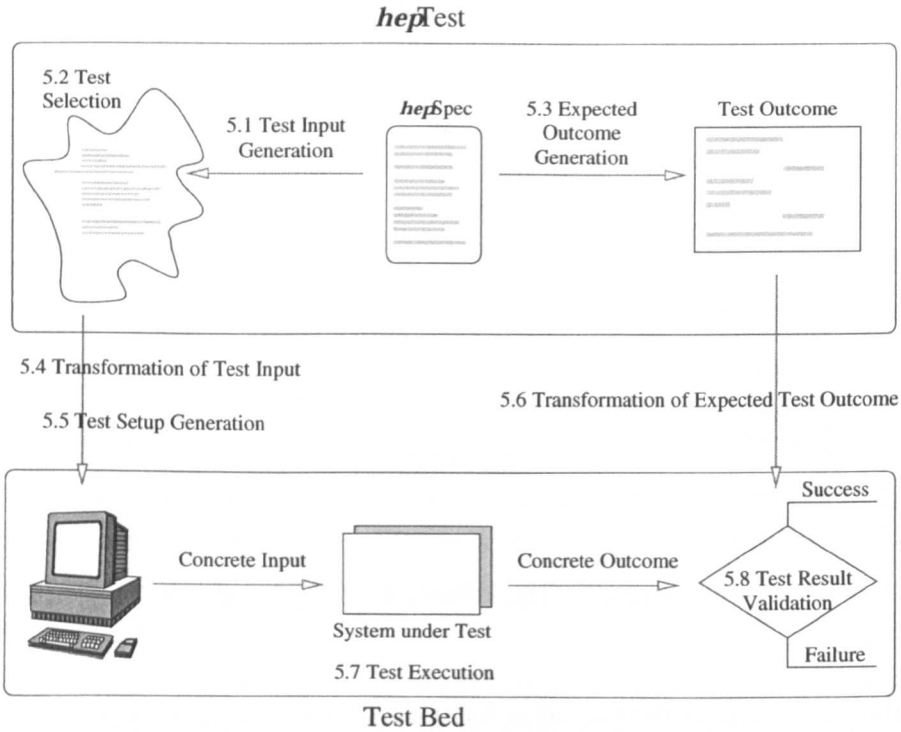


Figure 5.2: Combining Traditional Testing with *hep*SPEC— the *hep*TEST Approach

- Test input transformation
- Test setup generation
- Test execution
- Test outcome transformation
- Test result validation

The numbers in Figure 5.2 refer to the sections where each of the tasks is individually discussed.

Creating the specification is the initial step in the *hep*TEST-approach. An engineer has to write the specification defining the properties of the system under test which are of interest to him. We cannot automate this task as it is obviously the creative element. The engineer phrases informal requirements formally in the *hep*SPEC formalism. He can use a number of techniques to ensure that the specification describes accurately the desired properties of the system under test (Sommerville, 2001). Here we do not consider any validation techniques and assume that the specification is formally correct and the properties are defined correctly with respect to the requirements.

In the remainder of this chapter we will discuss each of the remaining tasks in the spirit of *hep*TEST.

5.1 Test Data Generation

The first step is the generation of test inputs. Inputs to a *hep*SPEC operation are terms of a specific sort. Thus inputs in *hep*TEST will be terms. We need to explain how inputs are created. We saw in Chapter 4 that ground terms denote values. Furthermore all values have a ground term representing them. This is important to automatic test case generation as it frees the engineer of having to choose a value during test generation. This notion gives rise to the idea of using ground terms as input values for testing.

We will define inductively what a test input of sort s is.

Definition 5.1.1 *Test input of sort s.*
Let $\Sigma = (S, F, arity, dom)$ be a hep-signature. Then the sets of strings $TI(\Sigma)_s$ for every $s \in S$ are called test input of sort s if they satisfy the following conditions:

- $\sigma \in TI(\Sigma)_s$ for every $\sigma \in G(\Sigma)$ with $\sigma : \rightarrow s$;

- if $\sigma \in G(\Sigma)$ with $\sigma\colon s_1, \ldots, s_n \rightarrow s$ and $\mathbf{t} = t_1, \ldots, t_n$ with each t_i a test input of the appropriate sort then $\sigma(\mathbf{t}) \in TI(\Sigma)_s$. ◊

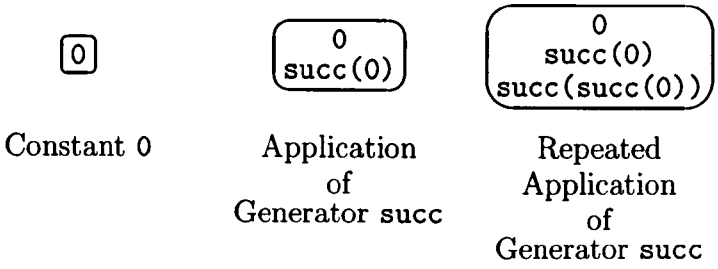


Figure 5.3: Test Inputs of Sort `Nat`

This definition states that all terms consisting of generators only are test inputs. Figure 5.3 illustrates the generation of test inputs of sort `Nat`. Starting with a set of constant generators the set of test inputs is enlarged by applying the `succ` operation of the elements of the previous set. The distinction between test inputs of different sorts stems from the notion that a term is of some sort s. We can say that test inputs of sort s are all ground generator terms of sort s. It is important to distinguish between test input of different sorts because we need to supply parameters to an operation of the appropriate sort. The application of an operation to values of the wrong sort is meaningless in the formalism.

5.2 Test Data Selection

In the following sections we will introduce formally our ideas for an order on test inputs. We will show some alternatives and discuss their advantages and disadvantages. Then we want to show the similarity between domain testing and *hep*SPEC-axioms. We explain our proposal for using axioms in domain analysis. We conclude the section on test selection by explaining ideas for support of traditional testing strategies and how they can be applied within *hep*TEST.

5.2.1 Order on Test Input

It seems that the sets of test inputs we can generate using the method above can be very large and even infinite. If a sort is defined recursively then the set of test inputs could be

infinite. There seems to be no way to stop the application of a recursive generator. We can observe this for example in Figure 5.4 where we could continue applying the operator succ indefinitely. In this case we would generate infinitely many test inputs. This is definitely true in the setting of total algebras. However in *hep*TEST we are able to restrict the application of recursive generators. In Chapter 4 we introduced domain restrictions for operators. We argued that many domains in practice are restricted. We can construct many examples were the domains seem to be unbounded. If we use for example natural numbers to describe a requirement of a system then we might encounter upper boundaries. Credit card numbers can serve as an example. Although is seems that credit card numbers can be arbitrary long, they are limited by the size of the physical credit card. A requirement is that the number has to fit on a credit card. Thus although we may use natural numbers to model credit card numbers the domain is not infinite. Quite on the contrary as it has an upper limit. The existence of this upper limit is essential to the model of credit card numbers and should be included into the specification. This is the case for most values in practice.

We can say that with domain restrictions we gain an upper boundary on domains for practical applications. Nevertheless the domains can still be very large. For practical reasons we need to select test inputs. But which ones? Looking back at the definition of test inputs we can observe that they are not an incoherent mass but have an order imposed on them.

Here we want to define a relation on ground terms which will prove to be extremely useful in testing: Ground terms occurring in other ground terms as subterms form a partial order over the set of all ground terms. Many strategies for test case selection require an order over values. We can make use of this order which is automatically given by the principle of term construction and does not need to be defined separately by the user.

Definition 5.2.1 *Immediate subterm.*
Let $\Sigma = (S, F, arity, dom)$ be a hep-signature and $t = f(t_0, t_1, \ldots, t_{n-1})$ a term over Σ. Every t_i of t is called an immediate subterm of t. We write $t_i \lhd t$. ◊

With this definition we can formulate an order over terms of a given sort s.

Definition 5.2.2 *Order of sort s.*
We define the predecessor relation $\overset{s}{\preceq}$ of test inputs of sort s as the reflexive, transitive closure of the immediate subterm relation $\lhd$ restricted to sort s. We will write $t1 \overset{s}{\preceq} t2$ if $t1$ is a predecessor test input of $t2$ in the predecessor relation of sort s. We will omit the s when it is obvious. ◊

It is obvious that the predecessor relation for a sort s forms a partial order over ground generator terms of sort s as it is reflexive, transitive and antisymmetric.

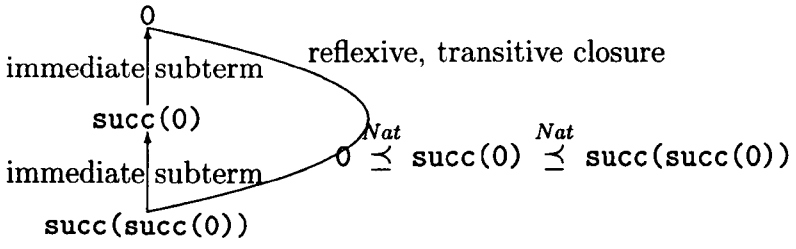


Figure 5.4: Partial Order for Terms of Sort `Nat`

In Figure 5.4 we can see how the test inputs of sort `Nat` which we generated in Figure 5.3 are ordered using the predecessor relation defined above. The arrows in the figure point to the predecessor term. We can observe that in this particular case the order is total and coincides with the familiar relation "less-or-equal" we might have used to order natural numbers. In Chapter 6 where we present case studies we will give more examples of ordering test inputs.

It is important to us that only terms of sort s are present in this order. When we select test inputs for an operation we need to be sure that they are of the appropriate sort. Consider Example 5.1 where we can easily see that $s(s(0)) \lhd app(empty, s(s(0)))$ holds

```
Seq is Nat with generation
 sorts Seq

 oprn empty → Seq
      app(Seq,Nat) → Seq
end Seq
```

$$s(s(0)) \overset{Seq}{\not\preceq} app(empty, s(s(0)))$$

$$s(s(0)) \lhd app(empty, s(s(0)))$$

Example 5.1: Difference between Immediate Subterm and Predecessor relation

because $s(s(0))$ is an immediate subterm of $app(empty, s(s(0)))$. Would we now select $s(s(0))$ as a test input where the system expects an input of sort `Seq` then the system would reject the input. We end up selecting a term which is a subterm but of the incorrect sort and create a problem when we try to apply the operation to the generated term of the inappropriate sort. This is why the definition of a predecessor relation on test input differs slightly from the definition of term order as used in term rewriting.

We have to look critically at the predecessor relation and discuss if it yields the results we expect in all of our settings. In Figure 5.5 we have drawn a fraction of the prede-

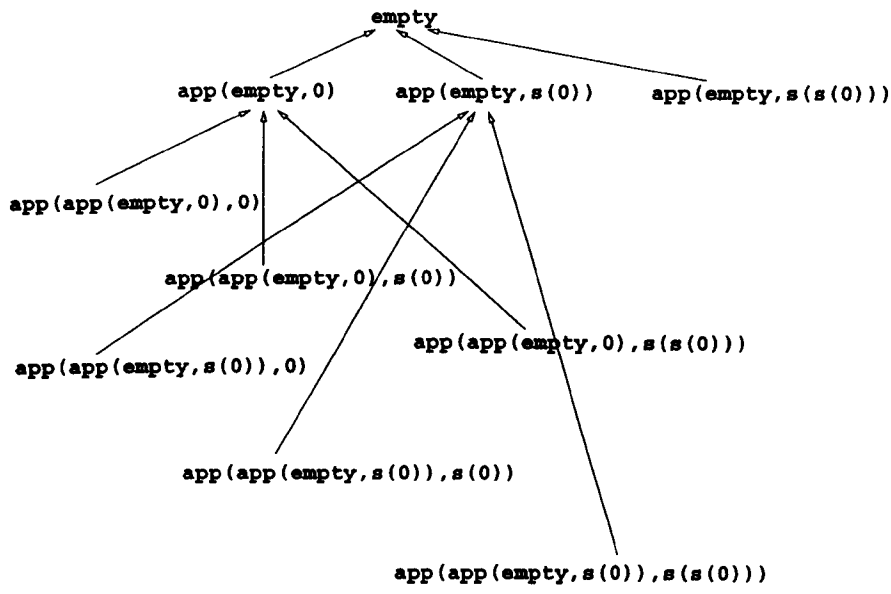


Figure 5.5: Partial Order on Seq

cessor relation for sort Seq taken from Example 5.1. The arrows in the figure point to the immediate sub-term of sort Seq. Because the predecessor relation does not define a total but only a partial order some terms cannot be compared. For example the term app(app(empty,0),0) cannot be compared with app(app(empty,0),s(0)) using the predecessor relation. If we encounter elements which we cannot relate to each other during test case selection, then we have to select both. In testing terms this means that the size of the test set grows and this might not be desirable. We want to look into another definition of a partial order which allows us to compare these two elements too. If we write the sequences like this 00 and 01 then we could argue that the first one is a predecessor of the second one. This would be the case if we try to find these sequences in a dictionary. There we would expect to find 00 before 01. However there are other reasons why the predecessor relation, as we defined it so far, is not practical for our needs.

The definition of the partial order we have used so far is useful for operations which take a single argument. There are many operations which have more. How do we define an order for a sequence of arguments? This questions needs answering not only to enlarge the set of comparable test inputs, as outlined above, but also since it impacts on test cases for defined operations. In Figure 5.5 we created part of the predecessor relation for sort Seq. Our specification could include an operation remove(Seq,Nat) $\rightarrow$ Seq which has 2 arguments. How do we select test cases for this operation? A quick fix would be to use

test inputs and the operation as a term to be ordered. Then `remove(app(empty,0),0)` could be considered a test input. Although the term is of sort `Seq` now, it does not provide the desired help for test selection. Firstly the term is not a generator term anymore, and we would need to change some of the previous definitions to reflect this fact. Furthermore there is another problem with this fix. It does not produce the desired result. None of the terms starting with `remove` is related to each other, because none is immediate sub-term of another. The operation `remove` is always the outer-most operation of such a term. This means that we have to select all terms as test cases regardless of the testing strategy. This contradicts our goals. That is why we decide to use a partial order defined on sequences of generator terms.

Definition 5.2.3 *Partial Order on Test Inputs.*
For each operation $\sigma\colon w \to s$ we define a relation $\overset{w}{\lhd}$ where for two term sequences of the appropriate sorts the following holds:

$$t1_1, \ldots, t1_j, \ldots, t1_n \overset{w}{\lhd} t2_1, \ldots, t2_j, \ldots, t2_n$$

iff for some $j \in 1 \ldots n$ $t1_j \overset{s_j}{\lhd} t2_j$ and $t1_i = t2_i$ for all $i \neq j$

As the partial order on test input sequences we define the transitive and reflexive closure of $\lhd$ which obeys the sorts. We write $\mathbf{t1} \overset{w}{\preceq} \mathbf{t2}$ to denote that term sequence $t1$ is a predecessor of term sequence $t2$. In the sequel we let $t1 \preceq t2$ denote $t1 \overset{w}{\preceq} t2$ when the tuple of sorts w is clear from the context. ◊

Note that we reused the relation symbols but that this time they are used over sequences of terms. The predecessor relation as defined earlier coincides with the partial order on test inputs for unary sequences. The order for `Seq` is illustrated in Figure 5.6 where the arrows in bold represent the additional members in the relation which now allows us to compare previously incomparable terms using the newly defined relation over term sequences. For example the operation `app` in Example 5.1 takes arguments of sort `Seq` and `Nat` and we use the order over tuples of sort `Seq,Nat`. The two terms we could not compare before can now be ordered. The term `app(empty,0),0`$\preceq$`app(empty,0),s(0)` because `0`$\preceq$`s(0)` holds.

To better understand the impact of this newly defined partial order consider the case of a defined operation like `add(Nat,Nat)`$\to$`Nat`. We can use the partial order over test input sequences `Nat,Nat` to select test cases for this operation. The elements we have to consider are still ground generator term sequences. Terms with defined operations are not part of

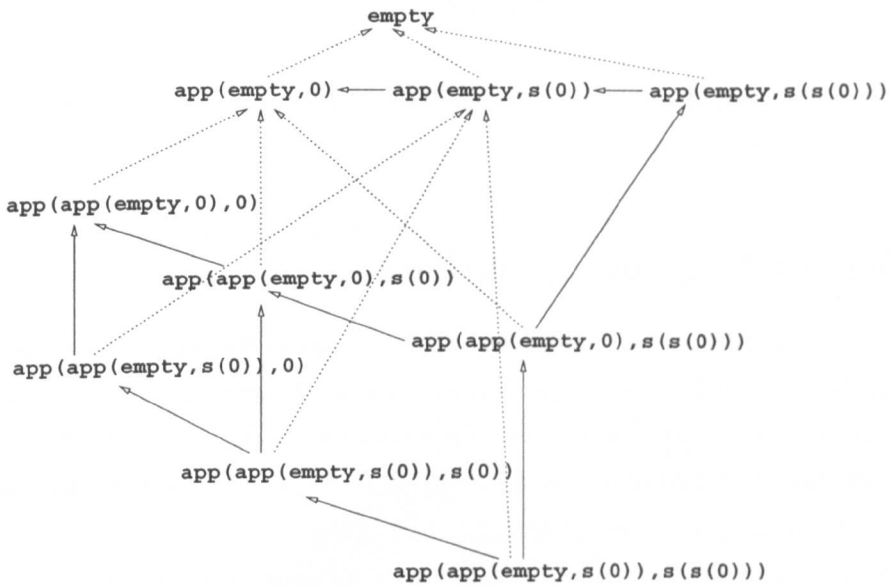


Figure 5.6: Partial Order for Test Inputs of sort Seq

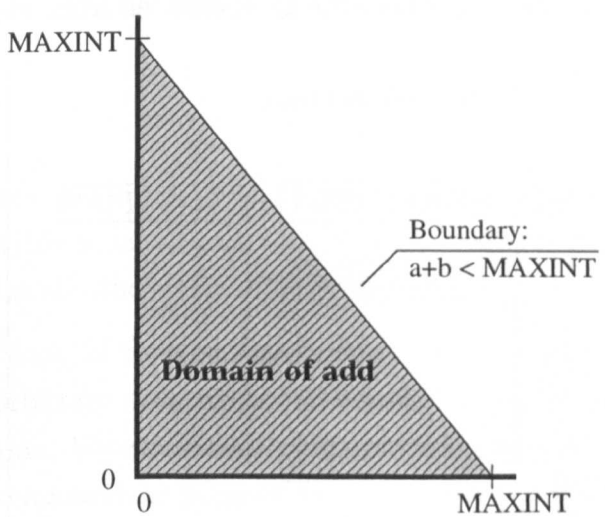


Figure 5.7: Plane of Natural Numbers

the order and we do not need to build equivalence classes as long as there are no axioms between generators. The partial order on `Nat,Nat` conforms nicely with our understanding of natural numbers. We can arrange the elements in a plane, using a familiar coordinate system as illustrated in Figure 5.7. Then we can observe that pairs $t_1 \overset{Nat,Nat}{\preceq} t_2$ are ordered according to the Euclidean distance from the point of origin.

5.2.2 Domain Selection by Axioms

In Chapter 3 we discussed the selection strategies in traditional testing approaches as advocated by Beizer (1995). Our main concern was that the strategy of boundary analysis was applicable only to rational numbers. Nevertheless it is important to note that the goal of finding boundaries or minima and maxima should not be restricted to numeric values only. What is hindering us to use it for arbitrary data structures?

In *hep*TEST we can deploy the same technology as used by Beizer in the case of rational numbers on any kind of data structure. To do that we need an order on the elements of a data structure. We have shown in the previous section how this is achieved. Furthermore we need means of identifying relevant domains and boundaries. Beizer's proposed strategy analyses a case statement to identify domains and sub-domains of operations. In *hep*TEST we use the domains created by the axioms of the operation. An axiom in *hep*SPEC has the general form of `if ...then ...` where the `if` is omitted when there is no premise.

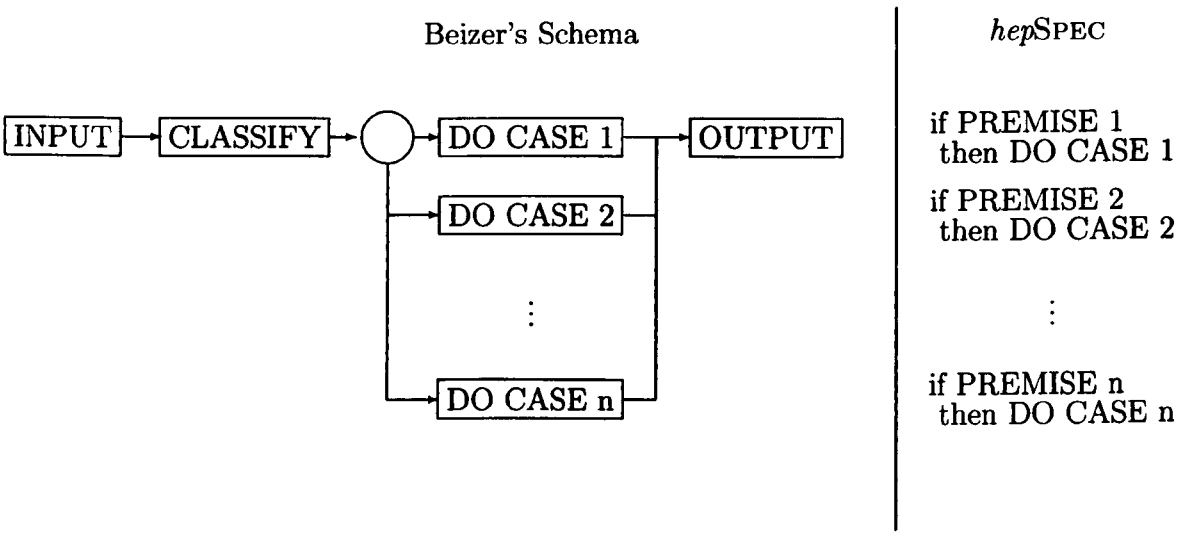


Figure 5.8: Domain Testing and *hep*SPEC

In Figure 5.8 we compare a schematic representation of domain testing with the approach in *hep*TEST. In contrast to Beizer's approach data structures considered in *hep*TEST can be arbitrary. Apart from that we can observe that the structure of a *hep*SPEC specification corresponds neatly with the schema representing the idea of domain testing. Thus we have lifted a traditional approach to the abstract level and are able to generalise it. In the same way as the case statement splits the input into sub-domains a set of axioms splits the domain of the operation defined by these axioms into sub-domains. We can associate a sub-domain with each axiom. The domain can be made visible in the test input order.

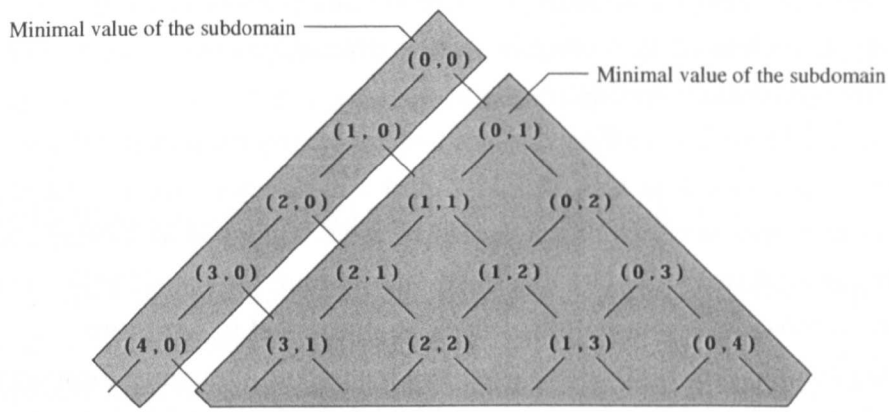


Figure 5.9: Domain Derivation from Axioms for Operation `add`

Figure 5.9 illustrates these ideas using the axioms for add as an example. The domain associated with `add(a,0)=a` contains all test inputs where the second argument is `0`. The second domain contains all other pairs of natural numbers.

As terms are the basis of tests in *hep*TEST we have the advantage that we can use domain testing with arbitrary data types. This is even stronger reinforced when we make the following observation. We can observe that the domains in *hep*TEST are not just sets. With the order on test inputs these domains become partially ordered sets. We can see from Figure 5.9 how the test inputs in the domains are ordered. The order on test inputs means that we are not limited to numbers like Beizer when we use domain testing strategies. This makes the *hep*TEST approach applicable to a wide variety of systems. This is a crucial observation when we decide which testing strategies we can support in *hep*TEST.

5.2.3 Testing Strategies

Beizer uses in his example the less-than relation on numbers to identify boundaries. We can construct an order automatically and can make use of it during test data selection in a similar way as Beizer does it for numerical values. The partial order over test inputs becomes the foundation of *hep*TEST testing strategies.

The existence of an order over test inputs helps us now to select tests which are relevant for a particular strategy. We need to remember that testing had a creative element in the traditional world. The test engineer selects tests which he thinks will expose bugs. In the *hep*TEST approach we rely on the experience of the test engineer. Rather than selecting individual tests he selects a strategy and the tedious work of creating tests is automated. Thus the creative element in testing is shifted from selecting individual tests to the task of finding the appropriate testing strategy.

In functional black-box testing domain testing strategies prevail (Beizer, 1995). Common to these approaches is that domains and sub-domains are identified. From these sub-domains test data is selected in a variety of ways. Bougé et al. (1986) uses for example a randomly selected test from the identified sub-domain. Beizer (1995) advocates the use of boundary values. All these strategies can be supported in *hep*TEST.

Following the ideas in Section 5.2.2 we can create sub-domains. Equivalence partitioning selects only a subset of the tests required by a boundary-based testing technology. The special points at the boundaries of sub-domains are considered relevant test inputs (Beizer, 1995). These special points are called ON-points and OFF-points. We introduced this technology in Chapter 3.4. In the context of *hep*TEST they become even more interesting. Using the partial order over test inputs we can select the minimal and maximal elements of the sub-domain. The assumption that an ON-point is the OFF-point of the neighbouring sub-domain (see Bingchaing Jeng, 1994) does not hold in *hep*TEST. This fact is a result of the existence of domain conditions we introduced in *hep*SPEC. The operation is not necessary total, it will in general be partial. In *hep*TEST we can select all the interesting ON-points and OFF-points by sorting the sub-domains according to our partial order and searching for elements which do not have a predecessor or successor in this partially order set. Those values are the relevant tests according to the boundary technique.

For example imagine an operation on natural numbers which does one thing for the numbers between 0 and 5, 10 and 15, and so forth and something else for 6 to 9, 16 to 19 up to 100. The case statement describing such an operation f would look like this:

$$f(i) = iff\ 0 \leq i \leq 100$$

```
Case i in
[0...5],[10...15],...,100: Do case1,
[6...9],[16...19],...,[96...99]: Do case2;
```

The relevant tests for case 1 using maximum-minimum strategy would be 0 and 100. The inputs 6 and 99 are the relevant test inputs for the second case. The equivalence classes following Beizer (1995) are the intervals separating the processing of i. The sub-domains are non-continuous in this case, meaning that there are gaps within a sub-domain. If we use ON-points and OFF-points we get the boundaries of the non-continuous sub-domains. The inputs are 0, 5, 10, 15, ... 100 for the first sub-domain and 6, 9, ... 99 for the second.

We observe that the value 101 is not included in this strategy. It would have been an OFF-point according to Beizer, but 101 is not part of the domain of the operation f. A strategy which does include such a test case is presented later in Section 5.9.

5.3 Generation of Expected Outcome

After the generation of test inputs and the selection of appropriate test sets we can focus on the generation of an expected test outcome.

Testers usually have to provide an oracle (Beizer, 1995) which will determine the result of a test run. In Chapter 2 we discussed how other approaches attempt to provide a solution to this problem.

When we introduced the testing process in Chapter 2.1 we mentioned the, so called, oracle problem. The oracle problem basically refers to the difficulties in test result validation. In literature the stack is often used as an example (e.g. Marre, 1995). There it is made clear that a program implementing a stack and an abstract description of it cannot be compared and found equal in general. In Marre (1995) a testing context is proposed in order to soften the problem. The testing context is a sequence of function calls which separate a complex object, like a stack, into its primitive components which then can be compared using trusted operations.

In *hep*TEST a practical view is taken. Because industry has gained experience in testing for decades and test tools, e.g. Visual Test, can perform test result validation which are satisfactory in most cases, we believe that for test result validation the generation of an expected outcome is crucial.

In *hep*TEST the generation of the expected outcome is possible. To create it we compute the result of the application of the operation to a selected test input. With the help of

the axioms in the specification we can determine the result. In order to automate the task the specification is animated in a tool. There is a variety of possibilities to animate algebraic specifications (Bouma and Walters, 1989). Usually a term rewriting system is used to symbolically execute the specification. Such a system has been used here.

In other words the computation of the expected outcome is done by solving the following equation:

$$op(testInput) = expectedOutcome$$

The result will be a ground generator term of the range sort of the operation under test.

We have to note that because we have the possibility to compute the expected outcome we have an advantage over methods where this computation is impossible or very complicated. Although it is not generally true that a solution of an equation can be computed we have made sure that it is true for any *hep*SPEC specification by restricting the operators to primitive recursion (see Chapter 4).

5.4 Transformation of Test Inputs

The *hep*TEST approach differs here from all the other approaches. We provide a mechanism by which test inputs are transformed into a concrete representation.

5.4.1 Syntactic Homomorphism

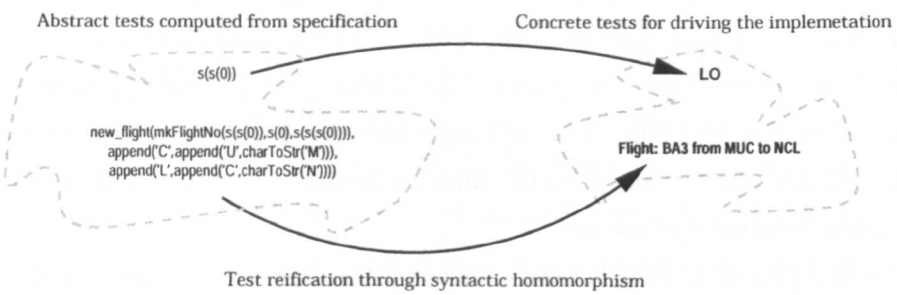


Figure 5.10: Test Input Transformation using a Syntactic Homomorphism

Figure 5.10 illustrates the problem and the solution provided in *hep*TEST. Test inputs generated from *hep*SPEC specifications are terms but the system under test is not capable

of processing these inputs. They have to be transformed into a representation the system or tester can deal with.

To transform test inputs we use a syntactic homomorphism to convert abstract values — ground generator terms — into concrete values which the system will process during test execution. In the spirit of *hep*TEST this conversion is automated to avoid the introduction of errors through manual conversion. The test inputs in *hep*TEST are difficult to read for an engineer. Here it would not matter if we choose infix, postfix or as we did prefix notation. It is the length of the terms and the possibly deep structure that causes the difficulties in comprehension. Just imagine the number 100 in Peano notation. Unarguably counting the s's manually would be error-prone. This is what we want to avoid while remaining within the formalism of algebraic specification.

To understand the advantages of using a single formalism we have to remember where test failures could originate. One source are requirements. A mistake in the requirements — be it omission, incompleteness, misunderstanding — can lead to an incorrect implementation of the system under test. Another source are bugs in the implementation. A test would fail because there is a discrepancy between the expected and the actual outcome. However we should not forget the third source of failure, the test itself. The test could be wrong in many ways. The inputs could be wrong and thus the expected outcome, but also the execution sequence might contain faults.

Test failure does not indicate the source of the fault. Finding the source of a fault is a laborious, time consuming and expensive task. Prevention of faults is therefore vital and using a single formalism helps us to ease the process of proving that a test is a logical consequence of a *hep*SPEC specification. An engineer might not use the proof method on a daily basis but its existence would make him confident to exclude one source of test failures.

In this setting we define a homomorphism that will achieve the transformation from abstract test inputs to concrete values. Formally we can say that our goal is to assign abstract terms or Σ-terms to another notation, i. e. we want to define another orthography. For that purpose we define another algebra I which contains all the elements of the concrete representation and associated string operators like concatenation. Then the homomorphism

$$syn\colon T(\Sigma) \rightarrow I$$

will achieve the necessary transformation.

In the spirit of algebras we can do the following. We can construct an algebra that describes the concrete representation. It would contain the characters of the concrete representation and string operators which combine these elements. Such an algebra contains a lot of values which could be described as nonsense. For example an algebra of arithmetic expressions would contain basic elements such as $+, -, *, \div$ and a representation for digits. Then values like $12 + +3 * - * *\div$ are nonsense in the world of arithmetic expressions. There is a subset of elements which have a meaning in the algebra generated by a *hep*SPEC specification of arithmetic operations. We want to illustrate this key idea. A Σ-algebra O

Specification of Σ	Σ-algebra O
Σ *is sorts* B,C,E *oprn* f,t → B 0,1 → E +,*(E,E) → E ∨,∧(B,B) → B ¬(B) → B κ(B,E,E) → E *end* Σ	Carriers: $O_E = \{a, b, c, d, [,], +, 0, 1\}^+$, $O_B = \{x, y, z, \wedge, \vee, \neg, [,], T, F\}^+$, $O_C = \{a, b, c, d, x, y, z, +, 0, 1, [,], \mathbf{IF}, \mathbf{THEN}, \mathbf{ELSE}, \neg\}^+$ Σ-indexed family of operations: $t^O = T, f^O = F, 0^O = 0, 1^O = 1,$ $(w_1, w_2)+^O = [w_1 + w_2],$ $(w_1, w_2)*^O = w_1 w_2,$ $(w_1, w_2)\vee^O = [w_1 \vee w_2],$ $(w_1, w_2)\wedge^O = [w_1 \wedge w_2],$ $w\neg^O = \neg[w],$ $(w_1, w_2, w_3)\kappa^O = \mathbf{IF}\ w_1\ \mathbf{THEN}\ w_2\ \mathbf{ELSE}\ w_3.$
v:{a,b,c,d,x,y,z}→{E,C,B} av = bv = cv = dv = E and xv = yv = zv = B	

Example 5.2: Syntactic Homomorphism $i\colon T(\Sigma, v) \to O$

with a different orthography is constructed in Example 5.2. We define the carrier set in O for sorts E, B, and C. They are non-empty sets over a given alphabet. We also provide definitions for string functions in O. For example the string function $+^O$ writes first the symbol [followed by the argument w_1, the symbol +, the second argument and the symbol]. Or for example κ^O concatenates strings using the arguments and the symbols **IF**, **THEN**, and **ELSE** interleaving them in a manner familiar in many programming languages. If we choose as assignment $i \in O_v$ the inclusion, then the extended homomorphism

$$i\colon T(\Sigma, v) \to O$$

associates terms of $T(\Sigma, v)$ with terms in mixed-fix notation.

It is important to note that not every string of the carrier in O is an image with respect

to the homomorphism $i\colon T(\Sigma, v) \rightarrow O$. Only those which have a meaning in Σ are assigned a value in O.

5.4.2 Grammars as Syntactic Homomorphism

To express the homomorphism we will use the formalism of context-free grammars (Aho et al., 1985). Context-free grammars are a well-known and well-understood formalism. Engineers make use of this formalism in the construction of compilers and related systems. We have used it too. In Example 5.2 we defined the operator κ^O as

$$(w_1, w_2, w_3)\kappa^O = \textbf{IF } w_1 \textbf{ THEN } w_2 \textbf{ ELSE } w_3.$$

We could have also used the formalism of grammars and written

$$E \rightarrow \textbf{IF } B \textbf{ THEN } E \textbf{ ELSE } E$$

in order to define κ instead. Before we get deeper into this discussion we will provide the necessary definitions.

Definition 5.4.1 *Context-free grammar.*
A grammar G is a quadruple (N, T, St, R), where:

- T is a set of terminals. The set of terminals is also called the alphabet.
- N is a set of nonterminals. Nonterminals are names that denote sets of strings.
- St is a distinguished nonterminal in N called the start symbol.
- R is the set of productions which determine how terminals and nonterminals can be combined to form strings. Each production consists of a nonterminal followed by an arrow ($\rightarrow$) and a string of terminals and nonterminals. The nonterminal on the left-hand side of a production is called the head, the right-hand side the body of the production. ◊

For the remainder of this thesis we will enclose a terminal in square brackets to distinguish it from nonterminals.

Another concept of grammars which is important to us is the notion of derivations. Derivation is a possible view of the process by which a grammar defines a language.

Definition 5.4.2 *Derivation.*
Let $A \rightarrow \gamma$ be a production of Grammar G and α and β are arbitrary strings of symbols from $N \cup T$ then we call $\alpha A \beta \Rightarrow \alpha\gamma\beta$ a derivation.

If $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ then we say α_1 derives α_n and write $\alpha_1 \stackrel{*}{\Longrightarrow} \alpha_n$. The relation $\stackrel{*}{\Longrightarrow}$ is the transitive and reflexive closure of $\Rightarrow$. If we want to express that a derivation takes at least one step then we use the symbol $\stackrel{+}{\Longrightarrow}$ (transitive closure). ◊

Definition 5.4.3 *Language.*
Let G be a grammar with start symbol St. Then $L(G)$ — the set of strings of terminals — is called the language generated by G. A terminal string w is in $L(G)$ if and only if $S \stackrel{+}{\Longrightarrow} w$ and w is called a sentence of $L(G)$. ◊

Parse trees are commonly used to describe the derivation process graphically. We will use them to illustrate our examples.

We said that a grammar generates a language. In *hep*TEST we rephrase this and say the grammar generates elements of the concrete representation. It becomes obvious that the elements of the generated language are a subset of the elements of the orthographical algebra as defined by Reichel (1987). However, unlike in Reichel (1987), in our algebra we do not generate elements which are syntactically incorrect. Those values are not in the range of the syntactic homomorphism and therefore we are only reducing the algebra of the concrete representation. In Section 5.9 we will explain how we can make use of this fact in other testing strategies.

```
1    Bin → [0]
2    Bin → [L]
3    Bin → Bin [0]
4    Bin → Bin [L]
```

Example 5.3: Grammar Productions Generating Binary Representation of Numbers

The algebra I of concrete inputs is generated by the grammar. To express the homomorphism h: $T(\Sigma) \rightarrow I$ from the abstract representation to the concrete we start by assigning values from $T(\Sigma)$ to the strings in I.

Definition 5.4.4 *Syntactic Homomorphism.*
Let Σ be a hep-signature and Γ a grammar describing the concrete data, then a homomorphism

$$syn: T(\Sigma) \rightarrow L(\Gamma)$$

is called a syntactic homomorphism. ◊

The recursion theorem in Reichel (1987, p. 68) points us to the way how to construct this homomorphism.

Definition 5.4.5 *Recursion theorem.*
Let $\Sigma = (S, F, arity, dom)$ be any hep-signature and X an S-sorted family of variables. For any Σ-algebra A and any assignment $\alpha = \langle \alpha_s \colon X_s \to I_s \mid s \in S \rangle$ there exists exactly one Σ-homomorphism $h \colon T(\Sigma, X) \to I$ with $h(x) = \alpha(x)$, i.e., every assignment can be extended uniquely to a Σ-homomorphism. ◊

As we consider ground-generator term algebras only (X is the empty family) this means that there is exactly one homomorphism from the algebra of ground-generator terms into a syntactic algebra I.

It remains to show how such a syntactic algebra I may be defined properly by means of a grammar Γ.

(1) At first any sort s of S is assigned a nonterminal of the grammar Γ.

(2) Every operator f of F is assigned a production p_i of the grammar Γ in the following way:

- The left-hand side of p_i is the nonterminal corresponding to the range of f.
- The right-hand of p_i is the sequence of nonterminals corresponding to the sorts s_i of f's input.
- This right-hand sequence of nonterminals is in general interspersed with keywords or special symbols denoting the operator f in $L(\Gamma)$.

(3) The carriers I_s of the syntactic algebra I are formed by all terminal strings derivable in the grammar Γ from the nonterminal corresponding to the sort s of S.

Let's now demonstrate with an example the definitions and processes just explained. In Example 5.4 we extend our hep-specification for natural numbers to `BinNat`. We introduce 4 new operators `zero`, `one`, `even`, and `odd`. Because this is an enhancement by definition we need to provide axioms for these operators. The axioms explain themselves. Then we extend our specification `BinNat` to build the syntactic homomorphism `Syn`. First we assign the sort `Nat` to a nonterminal `Bin` as explained above. Then follow the other operators which we assign grammar productions. The operator `zero` is assigned to the

```
1   BinNat is Nat with definition
2    oprn zero → Nat
3         one → Nat
4         even(Nat) → Nat
5         odd(Nat) → Nat
6    axioms a:Nat
7     zero = 0
8     one = succ(0)
9     even(a) = add(a,a)
10    odd(a) = s(add(a,a))
11  end BinNat
12
13  Syn is BinNat with
14   Nat means Bin
15   zero means Bin → [0]
16   one means Bin → [L]
17   even means Bin → Bin [0]
18   odd means Bin → Bin [L]
19  end Syn
```

Example 5.4: Definition of Syntactic Homomorphism for Binary Presentation of Natural Numbers

production `Bin → [0]`. We had to ensure that the left-hand side of the production is the non-terminal assigned to the range of `zero` which is `Nat`. The right-hand side of the production should contain the nonterminals of the domain of `zero` interspersed with terminal symbols. The domain of `zero` is empty and the right-hand side contains therefore only a terminal. Examine the assignments of `even` or `odd` as examples of assignments of operators with a non-empty domains.

We can observe that in order to write down the homomorphism in the proposed way we had to extend the original specification `Nat` to `BinNat` first. We could not have used `Nat` as it has only the operators `0` and `succ` and our grammar has 4 productions. The extension to `BinNat` serves only as a vehicle to express the homomorphism and does not add any new functionality to our specification. We can omit it and link the grammar directly to the original specification.

We can do it if we write the grammars in the usual way but add a keyword `means` to provide the link to the original specification. This link we call *means clause*. The Example 5.5 provides an illustration of how we express the homomorphism with the help of context-free grammars avoiding the use of additional operators. We can see that the axioms

```
1    Bin → [0] means Bin = 0
2        | [L] means Bin = s(0)
3        | Bin [0] means Bin = add(Bin1,Bin1)
4        | Bin [L] means Bin = s(add(Bin1,Bin1));
```

Example 5.5: Grammar Specifying a Homomorphism between Sort `Nat` and a Concrete Binary Representation

we wrote in the `BinNat` specification have been moved to the corresponding productions of the grammar. The first production for we associated with the operator `zero` in `BinNat` in Example 5.4 and we provided the axiom `zero = 0`. This axiom is now written behind the keyword *means*. To improve readability of grammars we use the nonterminals as variables in the means clause. To distinguish multiple occurrences of nonterminals we will number them starting with one as illustrated in the example. So the variable `Bin1` refers to the first occurrence of `Bin` in the body of the production. The nonterminal in the head of a production is not numbered. This is a convention to ease reading and writing of means clauses. The bar (|) is commonly used to denote alternatives in grammars (Aho et al., 1985) and the semicolon terminates the productions rules for `Bin`.

In this way we avoid to write unnecessary enhancements and therefore ease the writing and reading of specifications.

Definition 5.4.6 *Syntactic Grammar.*
Let $G = (N, T, St, R)$ be a grammar and $\Sigma = (S, F, arity, dom)$ a signature of a hep-specification. Then a syntactic grammar $SG = (\Sigma, G, Ax)$ has associated with each production a means-clause $eq \in Ax$. A means-clause is an equation eq over the non-terminals of the rule, where eq uses only operators in Σ. ◊

We have always emphasised the importance of the restrictions of the operators. Syntactic grammars as defined so far are ignoring these restrictions. The language generated by the grammar contains elements which do not have an abstract complement. We have already excluded elements of the orthographic algebra as defined by Reichel (1987) (see Example 5.2). We want to go even further. Following the concept of defining conditions in *hep*SPEC we will introduce restrictions on productions.

Definition 5.4.7 *Partial Syntactic Grammar.*
Let $SG = (\Sigma, G, Ax)$ be a syntactic grammar, then a partial syntactic grammar $PSG = (\Sigma, G, Ax, cond)$ has associated a where-clause with each rule. A where-clause contains a set of equations over the non-terminals of the right-hand side of the rule.

A production is only available for generation if the associated where-clause is satisfied.

◊

```
1    Bin → [0] means Bin = 0
2        | [L] means Bin = s(0)
3        | Bin [0]
4          where lt(0,Bin1) = true
5              leq(Bin1,127) = true
6          means Bin = add(Bin1,Bin1)
7        | Bin [L]
8          where lt(0,Bin1) = true
9              leq(Bin1,127) = true
10         means Bin = s(add(Bin1,Bin1))
```

Example 5.6: Partial Syntactic Grammar for a Byte Representation

In Example 5.6 we extended the grammar by providing where-clauses for grammar rules. Empty where-clauses were omitted. Now the grammar rules produce 8-bit binary representations for the natural numbers between 0 and 255 without leading 0's. We used the where-clauses to restrict the application of the grammar rules. In the same way as the domain conditions work in the case of hep-specifications we created a way to restrict the language generated by the grammar by using where-clauses. More importantly if the means-clause contains an operator with a restricted domain then we have the power to ensure that the operator is used within its domain through an appropriate where-clause.

5.4.3 Test Transformation through Reverse Parsing

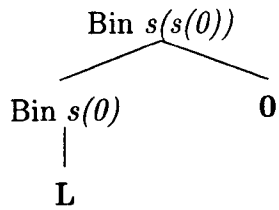
In order to automate the task of constructing the concrete input we will use the reverse parsing algorithm and the method described to calculate the expected outcome.

The process starts with the abstract test input as the semantic value of the root node of a parse tree. From the productions which would create the root node we select one. This production contains an action coded in the means clause. We can use it to calculate the semantic values for each child node. Then we proceed with the child nodes and continue this process until we reach the leaves of the parse tree. The sequence of terminals is then the concrete representation of the abstract test input.

Let us use the Example 5.5 again to illustrate the approach. We know that elements of `Nat` are represented by elements generated from the nonterminal `Bin`. So for a given test input we need to start the generation with the nonterminal `Bin`. From the grammar in

Example 5.5 we know that `Bin` can be expanded in 4 ways. We chose one arbitrary. Then we need to test if this is the right way. Using a term rewriting system we evaluate the means clause. If we have found a solution, then there are ground terms which need to be substituted for the nonterminals in the productions body, and we commit to the choice of this production. Our initial task is then multiplied. We need now to find representations for a number of test inputs. The procedure is the same as illustrated up to now. If the generation for all the nonterminals succeeds then we have found the concrete representation. The sequence of terminals in the parse tree is the concrete representation. Now it could have happened that the evaluation of the *means* clause does not yield a solution. Then another production is selected.

Let us use a concrete example, let us use test input `s(s(0))`. We illustrate the example



$\delta = (Bin \backslash s(s(0)))$
production 3: s(s(0)) = add(Bin1,Bin1)
$\delta = (Bin \backslash s(s(0)), Bin1 \backslash s(0))$
production 2: s(0)=s(0)
$\delta = (Bin \backslash s(s(0)), Bin1 \backslash s(0))$

Figure 5.11: Generation of Representation for s(s(0))

graphically in Figure 5.11. We start with the nonterminal `Bin` and select a production from the alternatives. Imagine we selected the third production (Line 3) then we have to evaluate the equation `s(s(0))=add(a,a)`. The equation holds if `a` is `s(0)`. We found a solution and infer from the body of the production that we first have to generate a representation for `s(0)` and append **0**. So we repeat the process this time for `s(0)`. We select a production, let us use the second production (Line 2 in Example 5.5). We evaluate the equation `s(0) = s(0)` and succeed. We add the terminal **L**. The process is finished and we have generated the concrete representation **L0** for `s(s(0))`. The tree we generated during the execution of the algorithm is a top-down created parse tree. Hence the name reverse parsing algorithm and reverse parse tree.

It is important to see what happens if we choose another production in the first step. Figure 5.12 shows the reverse parse tree in the initial step of transformation of the term `s(s(0))`. Instead of using the third production in the grammar (Line 3 in Example 5.5) we use this time the production in Line 4. We have to evaluate the means clause, here the

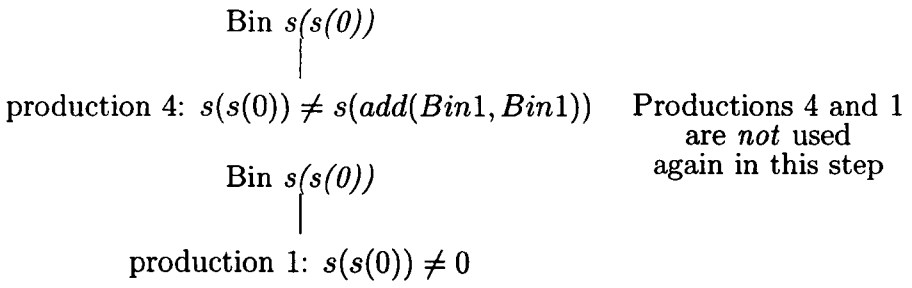


Figure 5.12: Generation of Representation for s(s(0)) with Failures

equation `s(s(0)) = s(add(a,a))`. We cannot find a solution so we discard this production in this step and select another one, say the first production (Line 1 in Example 5.6). Again we evaluate the means clause `s(s(0)) = 0` and fail to find a solution. This production is discarded in this step, too. This leaves only 2 productions to choose from. We know that if we choose the third production we will succeed in finding a solution for the means clause as we showed above. Choosing the third production means that we have to generate a `Bin` for the value `s(0)` as we explained in the previous example. Now all 4 productions for `Bin` are available for selection again. The process repeats until we find a production which fits.

It is possible that there is no solution which would yield the semantic values for the child nodes. Then we discard this production and use another production. If there is no other production, then we have to backtrack to the higher node.

In our formalism a test engineer will provide a syntactic homomorphism for test inputs in form of a grammar. This grammar will become an integral part of our specification. The test reification produces a test in concrete form from an abstract test input. This is achieved in *hep*TEST automatically through an algorithm that reverses the common parsing technique as described by Aho et al. (1985).

We followed some of the ideas of Bernot et al. (1991a) and a test of an operation is a pair of test inputs and an expected outcome. This idea was already presented in an earlier paper by Bougé et al. (1986). Woodward (1993) criticised this approach by comparing a test input with the actual input an implementation would accept. Then he concluded that such an approach only generates tests for specification validation purposes. His criticism is targeted at the fact that an implementation will not accept terms as input which is usually true. The *hep*TEST approach has overcome this problem.

5.5 Generation of Test Setup

Having used the reverse parsing technique for test input transformation we can extend this idea to generate the test setup. The test setup includes the operations and their input values which need to precede the actual test. From a viewpoint of parsing this task can be expressed as the search for a tree where the test input transformation parse tree is a sub-tree. Figure 5.13 provides a schematic illustration of this problem. The test setup

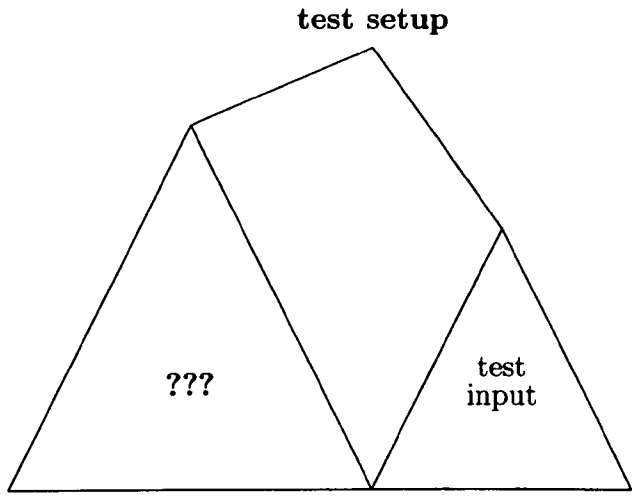


Figure 5.13: Test Setup Generation

generation is terminated once the start symbol of the grammar is reached.

Having expressed the problem of test setup generation in terms of grammars and reverse parsing we can explain how the solution in *hep*TEST is achieved. Let us recall that we used initially the syntactic homomorphisms to convert abstract test inputs into concrete representations. There we had to define a homomorphism between abstract ground generator terms and strings from a concrete representation. As a result we were able to convert automatically our abstract test input into a concrete representation. If we now extend this homomorphism to defined operations then we can generate in the same way the test setup.

In Figure 5.14 we have drawn a parse tree for a test setup. We assumed that there are buttons labelled with the terminal symbols. From our testing strategy we had generated the test input `(s(s(0)),s(0)` for the operation add. Then we used the reverse parsing algorithm to generate the binary representation for `s(s(0))` and `s(0)`. We continued to

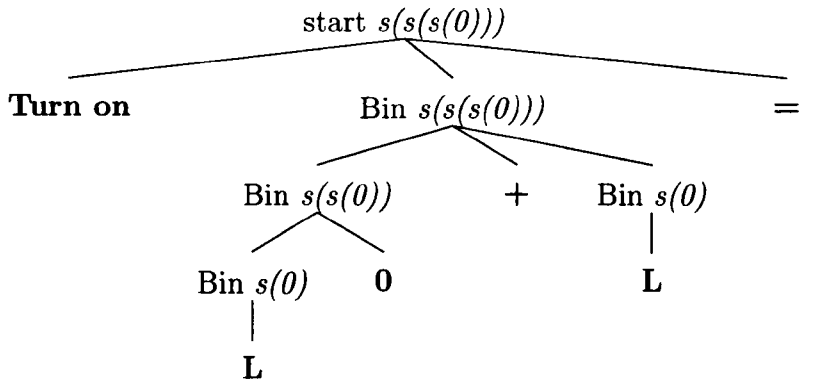


Figure 5.14: Test Setup Generation for Binary Calculator

create a reverse parse tree with the following additional productions until we reached the start symbol.

```
start → [Turn on] Bin [=] means start = Bin;
Bin   → Bin [+] Bin means Bin = add(Bin1,Bin2);
```

The result is the sequence `Turn on L 0 + L =` which we can enter into a binary desk calculator. These are very trivial examples, but in Chapter 6 we apply this technology to larger and more complex studies. There the power of the technology becomes more obvious.

Of course not all operations need to be transformed by this extended homomorphism. Some of the operations are auxiliary. They are only needed to express the defining condition of operators or to conveniently define the properties of another operation. Those auxiliary operations have no correspondence in the concrete world. We do not provide a homomorphism for them. Other algebraic methods introduce a hide operator to express this concept (Wirsing, 1990).

In Example 5.7 we illustrate the use of auxiliary operators. We also show that not only operators are auxiliary but also auxiliary sorts can be used. We define a sort **UniqueSeq** representing sequences of numbers which contains each number only once. Such a sequence could be used to describe properties of processes and process identifiers in computer operating systems, for example. Here we want to show the use of auxiliary operators. One of the generators of **UniqueSeq**, **append**, has a defining condition. We used an operation **makeSet** to express the condition. We can interpret it to mean that a number can only be appended if and only if the set formed from the sequence does not already contain the

```
UniqueSeq is Set with generation
 sorts UniqueSeq
with definition
 oprn makeSet(UniqueSeq) → Set
with generation
 oprn lambda → UniqueSeq
      append(s:UniqueSeq,n:Nat
        iff in(n,makeSet(s))=false ) → UniqueSeq
end UniqueSeq
```

Example 5.7: Use of Auxiliary Sorts and Operations

number. The implementation of UniqueSeq is unlikely to have an equivalent operation for makeSet. It will probably not create a set at all to ensure the property of uniqueness. In other words not only will there be no equivalent for makeSet but it is likely that neither the sort Set nor any operations associated with Set will be present in the implementation (unless it is needed for some other purpose in the system).

Thus the homomorphism is only defined for a subset of sorts and operators. The other operators can be called hidden. Here we follow a similar approach to LARCH (Wirsing, 1990). We will content ourself with this idea and omit the development of a fully fledged theory with hidden operations as we are able to overcome the problem of user intervention. In general only the tester will know which operations and sorts have an equivalent in the concrete representation. To enhance the possibilities of specification reuse we omit the explicit distinction of hidden operators.

5.6 Transformation of Expected Outcome

In a similar way the expected outcome is transformed using a context free grammar. It might be feasible to use the same homomorphism we used for test input and setup generation. However such a case would be the exception. In general we will make use of the same process but a different homomorphism. There is no reason why the inputs and outcomes — even thought they are of the same sort — will have necessarily the same concrete representation. It is easy to construct examples where this becomes obvious.

For example imagine a converter from numbers in binary to hexadecimal representation. There the input and the output obviously differ but they are both of sort Nat.

To distinguish between the homomorphisms we call the one which converts the expected outcome semantic homomorphism. This homomorphism is also expressed through

an extended context-free grammar in a similar way as syntactic homomorphism (see Section 5.4.2).

We can therefore distinguish three algebras in our context. First there is the abstract algebra or $T(\Sigma)$, then the input algebra I which we link to $T(\Sigma)$ through a syntactic homomorphism and third the output algebra O which we link to $T(\Sigma)$ through a semantic homomorphism. Figure 5.15 illustrates the links between the algebras. For our specifica-

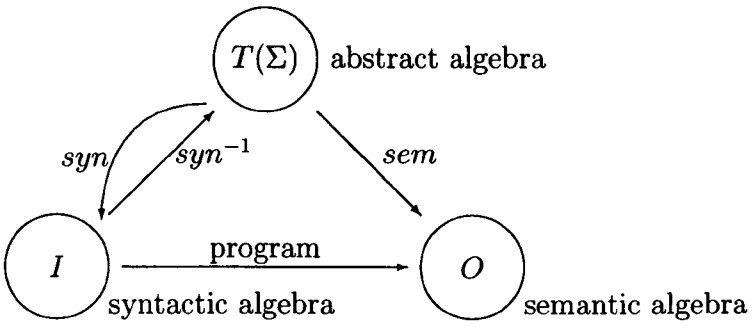


Figure 5.15: Links between Abstract, Syntactic and Semantic Algebras

tions it is important that the actual system is described through the concatenation of the inverse syntactic and semantic homomorphisms. The engineer writing the specification has the responsibility to ensure that this is true.

In the same fashion in which we generated a test setup we can use the grammar formalism to create any necessary context for the expected test outcome. Many of the ideas presented here will be exemplified in the case studies in Chapter 6.

5.7 Test Execution

The test scripts and outcome scripts need to be executed. This may be done automatically or manually. If the tests are executed automatically then automated testing software drives the process. Such software is programmed in a script language. There is no reason for us not to use this script language as the concrete representation when we specify the homomorphism.

There is a number of automated testing software available today. They all provide a script language to drive the software under test. Such software is used by testers to implement tests which they designed.

The *hep*TEST approach fits neatly into the ideas of automated test execution.

In other words, all we need to change when we move from manual to automated test execution is the definition of our syntactical homomorphism. The abstract tests are not affected by this. Merely their concrete representation will change. Also the effect which we talked about in the previous section can be used even more elaborately in this setting. The generation of test setup and sequencing becomes more important, because the launch of the system under test within the automated test execution bed might be more complicated than in the manual case.

Why is this an important advantage of *hep*TEST in comparison to other testing approaches? The advantage stems from the fact that *hep*TEST does use a homomorphism to link concrete and abstract test data. To the formal side of *hep*TEST the type of test execution is transparent. Actually *hep*TEST does not know if tests are executed manually or automated. Setting up an automated test environment can be expensive. In the early stages of software development the system might be unstable, meaning it will fail in a larger number of tests. If the test execution is performed automatically and the testing stops after the third test already then the fixing, restarting and repeated failure will slow down an automated process. Manual testing might be preferred in this case. However later when bug fixes take place the same abstract tests can be run in an automated environment. The number of test failures should be small and not contributing to a slowdown of the testing effort. Because we only change the homomorphism we can be confident that the same tests are executed and that regression testing will yield a high level of confidence in the systems correctness.

5.8 Test Result Validation

The contribution of *hep*TEST to the test result validation process is significant. Earlier we argued that an expected outcome is needed in order to establish test success or failure.

We pointed out that many approaches are not capable to produce an expected outcome. The approaches using object oriented description methods such as UML are not capable of producing an expected outcome because the descriptions do not contain any semantics. They usually argue that the expected outcome can be derived from another source. This source might be a legacy system which is to be replaced by the system under test or some other kind of model. From a practical point of view it means to rely on the fact that the legacy system is correct or the the model is adequate and in sync with the specification.

Nothing of this is generally true.

We also argued that in the case of manual ad-hoc testing the tester usually knows what to expect as an outcome. This is not longer true in the case of automated test generation. A tester would need to trace the test input and analyse the effects to determine successfully the expected outcome. This is a very time consuming and unreliable method.

In *hep*TEST we did provide semantics in our specifications. The semantics are formal and can be manipulated mechanically to compute the expected outcome. We can also compute the expected outcome for testing sequences and — using the transformation approach — convert the representation of the expected outcome into a format where the test result validation is eased.

The syntactic homomorphism can be extended in such a way that not only the expected outcome is transformed but also the necessary calls and actions to retrieve it are included. In this way we can support automated test result validation.

5.9 Further Testing Strategies

In Section 5.2.3 we discussed the use of *hep*TEST for functional black-box testing where the domain-based testing strategies dominate. With the ideas presented so far we are able to provide support in two other areas of testing, namely robustness and syntax testing.

Automated test generation for robustness testing is presented for example by N.P.Kropp et al. (1998). The goal of robustness testing is to break the software, to stop it from functioning. Test result validation has to establish if the software under test is still running. Validation of the outcome is not important. In *hep*TEST we can define tests for robustness testing as arbitrary strings of terminals for the syntactic grammar. Formally if $G = (N, T, St, P)$ is a syntactic grammar then a robustness test t is an element of T^+.

Consider again our example of a test setup in Figure 5.14. There robustness tests of binary numbers and the addition could be strings like **+LL+LL+++L00**.

Another testing approach is discussed for example in Beizer (1995) and called syntax testing. We could argue that the test engineer is aware of the fact that the software or system is designed and constructed which does not permit or avoids problems associated with input syntax. Such technology includes parser generators, generated lexical analysers for example. This technology does the reverse of our syntactic homomorphism. A parser is constructed to automatically decide if a sequence is a sentence of a given language. When automated construction tools are used on the project then certain tests can be

omitted. The testing strategy should concentrated more on the remaining sources of bugs. For reasons of time and space efficiency only context-free grammars are of any practical importance. This in turn means that the remaining sources of faults are located in the implementation of context-sensitive checks. If we compare this approach to *hep*TEST then we can say that syntactic testing should focus on testing the where-clauses in the syntactic grammar.

To test the implementation of the where-clauses we can proceed in a similar way as we introduced for domain testing in Section 5.2.3. If we can identify the domains of rules, then we can use the partial order on test inputs to select specific tests. Using the domain testing approach we already have tested a considerable number of values. To be more specific at finding faults specific to syntax testing, we would consider to test if the domains of the where-clauses are implemented correctly. So the first type of syntax tests would be

$$t \in L(G) \text{ is a syntactic but not necessarily context-sensitive correct test.}$$

The second class would be

$$t \in L(G) \cap L(EG).$$

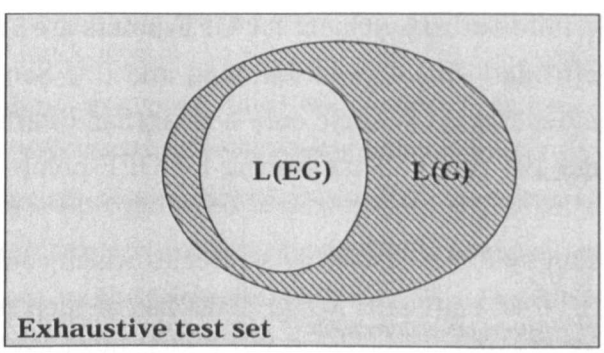


Figure 5.16: Syntax Testing Approach

Using the illustration in Figure 5.16 we can compare the potential test sets of these two classes of syntactic tests. We see that the elements of $L(EG)$ might have already been selected by the domain testing approach. The where-clause implementation would be best tested on the boundaries. The ON-points of the boundaries of the where-clause where already tests in $L(EG)$ thus OFF-points should be selected for the second approach.

Using the partial order we can select tests just outside the boundary of the where-clause. Formally we say a test input t tests the where-clause eq

$$t \notin \text{solution of } eq,\ t_{on} \lhd t \text{ or } t \lhd t_{on},\ t_{on} \in \text{solution of } eq.$$

```
1    ⋮
2        | Bin [0]
3          where lt(0,Bin1) = true
4              leq(Bin1,127) = true
5          means Bin = add(Bin1,Bin1)
6    ⋮
```

Example 5.8: OFF-point Testing in Syntax Testing

Using again Example 5.6 we want to select a syntactic test for testing an OFF-point for a where-clause. We copied the relevant part of the grammar to Example 5.8. We can see that the domain of the production rule (in Line 3 and 4) is described by the following inequality:

$$0 < Bin1 \leq 127$$

The tests which would fulfil our requirement for OFF-points are `Bin1` = `0` and `Bin1` = `128`. This is because `0` $\lhd$ `s(0)` and $s(0) \in$ Solution of eq and $0 \notin$ Solution of eq. The same is true for `128`. The number `128` is of course only a short-hand for a term in Peano notation where the `s` is repeated 128 times. Thus testing for OFF-points would yield 2 test cases for this production.

These are only a small number of testing strategies which could be implemented using the *hep*TEST approach. Test engineers would make use of their experience and formulate their own testing strategies. Our approach is there to help to put testing on a more formal basis. It does not substitute the creative element of testing. In the next chapter we will report about two case studies we conducted to assess the feasibility of this approach.

Chapter 6

Case Studies

In this chapter we will demonstrate the practical application of the techniques developed in the preceding chapters. The first example is a simple and well known one from the literature. It is used to test our ideas and their implementation. It small and relevant in our context and therefore ideal for an initial assessment of the proposed technology. This simple example is followed by a larger case study supplied to us by Dr Jeremy Dick. We conclude with a report of a real industrial case study in which a development project was "shadowed" and *hep*TEST approach tried out in a "real world" setting. We will report our experiences and findings.

6.1 Initial Case Study — Tax Example

The first case study has been conducted to assess the validity of our approach to testing. We also need to examine how the tools implementing the approach might cooperate and and gain some confidence in their implementations. We choose to use an example from Beizer (Beizer, 1995) because it is relatively small and we can compare the results with the results generated manually by Beizer. We will follow the major steps in testing as outlined in Chapter 5 beginning with modelling of the system under test.

6.1.1 Specification of the Tax Example

The tax example is taken from Beizer (1995) where he demonstrates the technique of domain analysis. The system is required to calculate the amount of income tax payable under US law. The tax system is progressive and the percentage increases in steps over

the income span. Figure 6.1 gives an indication of how a plotted graph of a function *calcIncomeTax* could look.

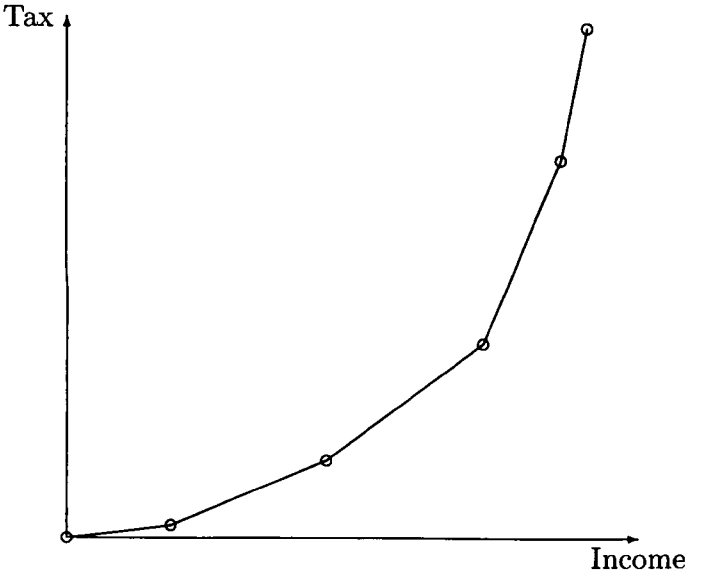


Figure 6.1: Graph for Tax Example

We can observe that there are bends in the graph at certain points. Beizer points out that the values where the graph makes those bends are potential test cases and we agree with that. Our goal therefore is to generate test cases for those and to take surrounding values into account.

We begin by formalising the system in *hep*SPEC. Firstly we create some basic sorts and operators which we will need to describe the properties of the example. Example 6.1 introduces the sorts `Bool` and `Nat` and operations to manipulate the elements.

It is important to point out that the operations `true` and `false` are the *generators* for sort `Bool` in the same way as `s` and `0` for `Nat`. The distinction between defined operators and generators is important as pointed out in Chapter 4. The generators are constructing the elements from which we will choose the test inputs.

From the basic specifications we can construct our example description. In Example 6.2 we define an operation `calcIncomeTax` which operates on elements of `Nat`. This example is based on the idea that we could represent a money value as a natural number. We did not follow the exact specification of income tax calculations because we used the Peano notation for natural numbers and this makes specifying numbers such as 22000 very tedious.

```
1   Bool is generation
2     sorts Bool
3     oprn
4        true → Bool
5       false → Bool
6   end Bool
7
8   Nat is Bool with generation
9     sorts Nat
10    oprn 0 → Nat
11         s(Nat) → Nat
12    with definition
13    oprn add(Nat,Nat) → Nat
14         mul(Nat,Nat) → Nat
15         leq(Nat,Nat) → Bool
16         sub(Nat a,Nat b iff leq(b,a)=true) → Nat
17    axioms a,b:Nat
18      add(a,0) = 0
19      add(a,s(b)) = s(add(a,b))
20
21      mul(a,0) = 0
22      mul(a,s(b)) = add(a,mul(a,b))
23
24      leq(0,b) = true
25      leq(s(a),0) = false
26      leq(s(a),s(b)) = leq(a,b)
27
28      sub(a,0) = a
29      if leq(b,a)=true
30       then sub(s(a),s(b)) = sub(a,b)
31  end Nat
```

Example 6.1: *hep*SPEC-specification of Basic Sorts for Tax Example

For an initial study this is an accepted restriction. We will discuss ideas to overcome this problem later in this chapter.

```
1  Tax is Nat with definition
2    oprn calcIncomeTax(Nat a
3      iff leq(a,s(s(s(s(s(s(s(s(s(s(0)))))))))))=true ) → Nat
4    axioms income: Nat
5      if leq(0,income) = true
6         leq(income,s(s(0))) = true
7        then calcIncomeTax(income) = mult(income,s(s(0)))
8
9      if leq(s(s(s(0))),income) = true
10        leq(income,s(s(s(s((0)))))) = true
11       then calcIncomeTax(income) =
12             add(mult(sub(income,s(s(0))),s(s(s(0)))),s(s(s(s(0)))))
13
14     if leq(s(s(s(s(s(0))))),income) = true
15        leq(income,s(s(s(s(s(s(s(s(0)))))))) = true
16       then calcIncomeTax(income) =
17             add(mult(sub(income,s(s(s(s(s(0)))))),s(s(s(s(0))))),
18                 s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))
19
20     if leq(s(s(s(s(s(s(s(s(0)))))))),income) = true
21        leq(income,s(s(s(s(s(s(s(s(s(s(0)))))))))) = true
22       then calcIncomeTax(income) =
23             add(mult(sub(income,s(s(s(s(s(s(0)))))),
24                       s(s(s(s(s(0)))))),
25                  s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
26                   s(s(s(s(s(s(0))))))))))))))))))))))))
27 end Tax
```

Example 6.2: *hep*SPEC-specification of Tax Example

To be able to write it conveniently and to be able to focus on the task at hand be have changed the specification slightly. The *hep*SPEC-specification of `calcIncomeTax` can be given by the following formula:

$$calcIncomeTax: \quad [0,\ldots,11] \rightarrow \mathbb{N}$$

$$calcIncomeTax(x) = \begin{cases} 2x & \text{if } x \leq 2, \\ 3(x-2)+4 & \text{if } 3 \leq x \text{ and } x \leq 5, \\ 4(x-5)+13 & \text{if } 6 \leq x \text{ and } x \leq 8, \\ 5(x-8)+25 & \text{if } 9 \leq x \text{ and } x \leq 11. \end{cases}$$

Having formalised the system under test we proceed to the generation steps. In the remainder of this discussion we will follow the structure of Chapter 5 and show how each step of the *hep*TEST process is applied in practice. The first one concerns the generation of test data.

6.1.2 Test Data Generation for Tax Example

The operation `calcIncomeTax` describes our entire system. Other operations mentioned in the specification such as `add` or `mul` are regarded as auxiliary. Thus our focus for test data generation is on `calcIncomeTax`.

From the specification in Example 6.2 we can see that we need to generate values of sort `Nat`. The sort `Nat` has 2 generator operators `s` and `0`. Starting with the operator `0` and repeated application of `s` we can generate all values of sort `Nat`.

To automate this task we can use a variety of tools. We could for example write a simple program which would generate the values. However we chose a more general approach as illustrated in Figure 6.2 by using a term rewriting system because it fits neatly into the remaining test data generation process. The implementation is largely based on the LoFT-tool(Marre, 1995) and uses the underlying Prolog engine from ECL^{i}PS^e (Aggounand et al., 1999) directly for extensions. We will continue to present this figure and to illustrate which parts of the implementation are relevant to a particular test generation step. Our implementation supporting the *hep*TEST approach is called hepTool. The hepTool is implemented in Java and provides a graphical user interface to the test engineer. A screen shot of the tool is presented in Figure 6.3.

In this example test input for `calcIncomeTax` generation is trivial. Nevertheless we need to select test cases from those we can generate because there is an infinite number of natural numbers.

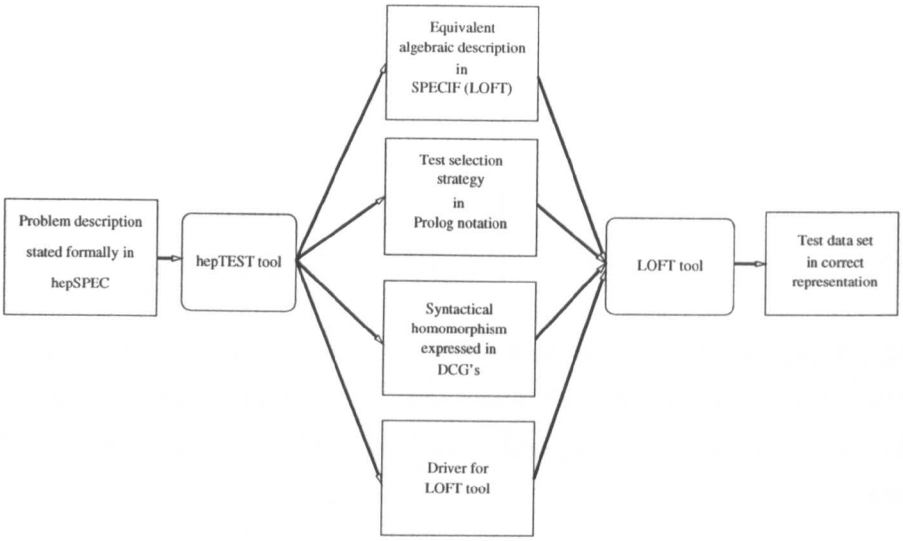


Figure 6.2: An Implementation of *hep*TEST

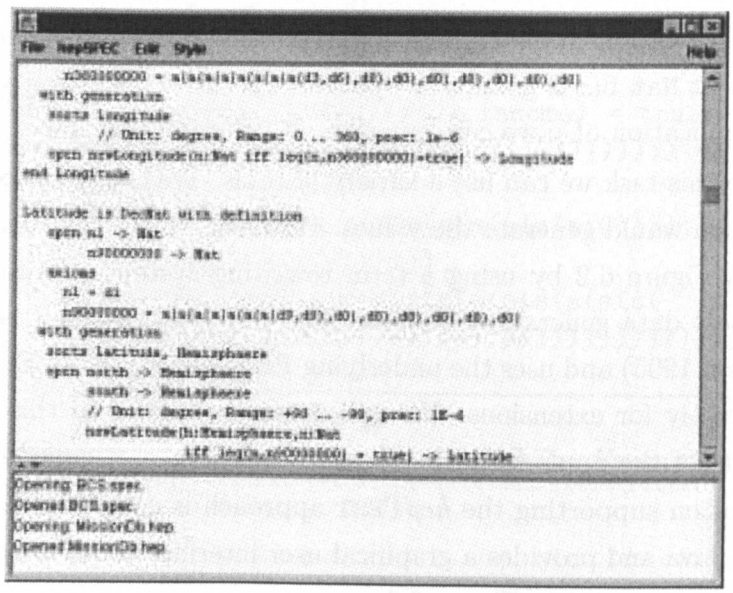


Figure 6.3: The hepTool User Interface

6.1.3 Test Data Selection for Tax Example

In its structure the specification reflects the ideas put forward by Beizer (1995) before he introduces this example. He considers that a system could be viewed as a huge case statement, where each of the cases is associated to a process that will operate on the given data. The derived testing strategy is concerned with finding faults in the implementation of such a case statement. In this section we will discuss the generation of test cases according to this particular strategy.

Our foremost goal was to assess the test generation potential with a focus on practical testing strategies. Thus we used the boundary analysis technique as demonstrated by Beizer (1995) for this study.

First we have to identify the domains for the system as presented in the Example 6.2 earlier. In this example it is easy to identify the domains as they are given in the description explicitly. Our goal is to replicate the process automatically.

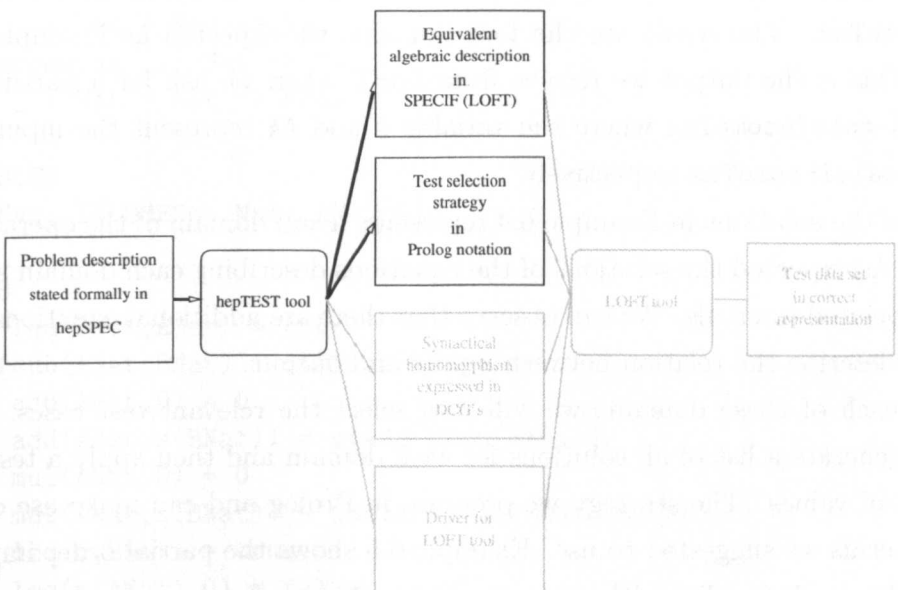


Figure 6.4: Introducing LoFT into *hep*TEST

We will use the capacities of the LoFT tool to identify the domains (Marre, 1995). However we need to take some additional steps because LoFT does not handle *hep*SPEC-specifications directly. First we transform the *hep*SPEC description into a specification which is understood by the LoFT tool. This step is illustrated in Figure 6.4. It shows the implementation we chose to use and highlights the particular parts of the setup which are

used in test generation and selection steps.

We also need to flatten the specification, reorganise and modify it to accommodate the LoFT tool conventions. This is done automatically by the *hep*TEST tool. The partial result of the transformation of Example 6.2 can be seen in Example 6.3.

There are additional operators and axioms to provide support for the operators with a restricted domain from *hep*SPEC. We need to guard operations with a restricted domain from being applied to erroneous values explained in Chapter 4.4 when we introduced hep-specifications. So for each occurrence of the operator restricted operator `sub` and `calcIncomeTax` in the axioms we add an equation ensuring that the restriction is obeyed. For example in line 25 we provide the definition for the auxiliary operator `defsub` which returns `defTrue` if the domain restriction of `sub` holds. This operator is then added to the conditions in the axioms where `sub` is used. This information can be extracted during syntax analysis and is therefore automated.

In the next step we use the LoFT tool to extract the domains for the operation `calcIncomeTax`. The result are the four domains we expected as Example 6.4 demonstrates. This is the output we receive from LoFT when we ask for a partitioning of the domain of `calcIncomeTax` where the variable `A` and `AA` represent the input and output values of `calcIncomeTax` respectively.

Each of the solutions in Example 6.4 represents a sub domain of the operation `calcIncomeTax`. As expected the solutions of the equations describing each domain are the values in the associated intervals. We can observe that there are additional equations e. g. in Line 13. They describe the relation between inputs and output.

From each of these domains we will now select the relevant test cases. We can ask LoFT to generate a list of all solutions for each domain and then apply a testing strategy to the list of values. The strategy we program in Prolog and can make use of the partial order on terms we suggested to use. Example 6.5 shows the partial order implemented in Prolog. The result is a list with relevant test inputs.

6.1.4 Test Outcome Computation for Tax Example

The selected inputs are used to generate an expected outcome. In our case the computation of an expected outcome is easy. We took the necessary precautions when we created the theory in Chapter 4.

With a term rewriting system as an implementation basis we can use its capabilites to compute the expected outcome for each test. All we need to do is to solve the equation

```
1  SPECIF Tax
2
3  SORTS
4  defBool Nat Bool
5
6  GENERATORS
7  defTrue: -> defBool
8  true: -> Bool
9  0: -> Nat
10 s _ : Nat -> Nat
11 false: -> Bool
12
13 OPERATIONS
14 defsub _ _ : Nat * Nat -> defBool
15 sub _ _ : Nat * Nat -> Nat
16 add _ _ : Nat * Nat -> Nat
17 mul _ _ : Nat * Nat -> Nat
18 calcincometax _ : Nat -> Nat
19 leq _ _ : Nat * Nat -> Bool
20
21 VARIABLES
22 BNat:Nat, INCOMENat:Nat, ANat:Nat
23
24 AXIOMS
25 Tax1: leq(BNat,ANat) = true
26 => defsub(ANat,BNat) = defTrue
27 Tax2: add(ANat,0) = 0
28 Tax3: add(ANat,s(BNat)) = s(add(ANat,BNat))
29 Tax4: mul(ANat,0) = 0
30 Tax5: mul(ANat,s(BNat)) = add(ANat,mul(ANat,BNat))
31 Tax6: leq(0,BNat) = true
32 Tax7: leq(s(ANat),0) = false
33 Tax8: leq(s(ANat),s(BNat)) = leq(ANat,BNat)
34 Tax9: defsub(ANat,0) = defTrue  => sub(ANat,0) = ANat
35 Tax10: defsub(s(ANat),s(BNat)) = defTrue
36 & defsub(ANat,BNat) = defTrue
37 & leq(BNat,ANat) = true => sub(s(ANat),s(BNat)) = sub(ANat,BNat)
38 ...
```

Example 6.3: Result of Converting the Example for Usage in LoFT

```
1    FINAL BINDING:
2    REMAINING CONSTRAINTS = {
3      leq(A:Nat,s(s(0))) = true,
4      mul(A,s(s(0))) = AA:Nat
5      }
6    SOLUTION #1,      CPUTIME = 10
7
8    FINAL BINDING:
9    REMAINING CONSTRAINTS = {
10     leq(s(s(0)),A:Nat) = true,
11     leq(s(s(s(0))),A) = true,
12     leq(A,s(s(s(s(s(0)))))) = true,
13     sub(A,s(s(0))) = _v0:Nat,
14     mul(_v0,s(s(s(0)))) = _v1:Nat,
15     add(_v1,s(s(s(s(0))))) = AA:Nat
16     }
17   SOLUTION #2,      CPUTIME = 0
18   :
19   GLOBAL TIME ELAPSED = 29
20   NUMBER OF SOLUTIONS = 4
```

Example 6.4: Domain Descriptions in LoFT for Tax Example

```
1    is_of_sort_NAT('0:->Nat').
2    is_of_sort_NAT('s:Nat->Nat'(X)) :- is_of_sort_NAT(X).
3
4    im_subterm_of_sort_NAT(X,'s:Nat->Nat'(X)) :- is_of_sort_NAT(X).
5
6    po_on_NAT(X,X) :- is_of_sort_NAT(X).
7    po_on_NAT(X,Y) :- im_subterm_of_sort_NAT(X,Y).
8    po_on_NAT(X,Y) :- X~=Z,im_subterm_of_sort_NAT(Z,Y),po_on_NAT(X,Z).
9
```

Example 6.5: Partial Order for Sort nat in Prolog

`function(input)=outcome`. We observed when discussing Example 6.4 that this output can be computed through the use of LoFT. So we use exactly this approach and insist that the solution is given explicitly rather as an equation system describing the solution.

6.1.5 Test Transformation for Tax Example

The results of the process so far are abstract tests which then need to be transformed into actual tests using the syntactic and semantic homomorphisms respectively.

We proposed in Section 5.4.2 and Section 5.6 to use grammars to express those homomorphisms. Having chosen a Prolog-based implementation we note that context-free grammars can be expressed as declarative clause grammars (Sterling and Shapiro, 1986) or DCGs for short. DCGs are usually build into the Prolog engines and ECL^{i}PS^e has them too. Figure 6.5 shows an updated view of our tool setup after the introduction of DCGs. The DCGs are extracted from the hep-specification.

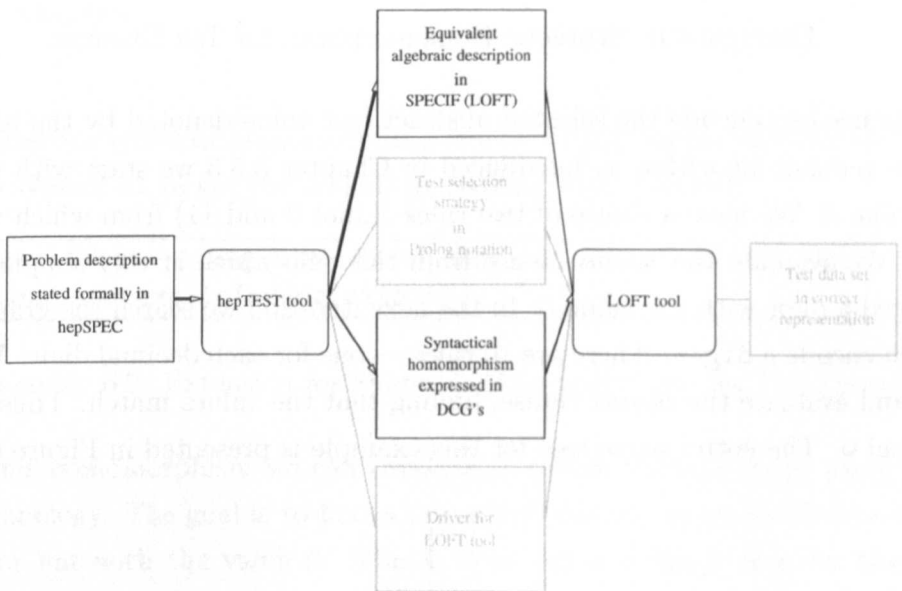


Figure 6.5: Implementation of Homomorphisms in *hep*TEST

To illustrate how the homomorphisms are utilised for the transformation we will go into details for a particular input. The value `0` is the minimum for the first domain. The expected outcome is `0` as well. Example 6.6 shows the syntactic homomorphism used in this example expressed as a grammar. We will start the generation process using `nat` as the

start symbol because nat encodes the transformation of the input data for the operation calcIncomeTax.

```
1     digit: [0]
2       means digit=0
3           | [1]
4       means digit=s(0)
5           :
6           | [9]
7       means digit=s(s(s(s(s(s(s(s(s(0)))))))));
8
9     nat: digit
10      means nat=digit
11       | nat digit
12      means nat=add(nat1,add(nat1,add(nat1,add(nat1,add(nat1,add(nat1,
13                add(nat1,add(nat1,add(nat1,add(nat1,
14                digit)))))))));
```

Example 6.6: Syntactic Homomorphism for Tax Example

Now we need to encode the selected abstract test value denoted by the term *0*. Using the reverse parsing algorithm as introduced in Chapter 3.5.3 we start with the root **nat** and the value *0*. We have a choice of two rules (Lines 9 and 11) from which we select the first rule. We evaluate the *means* clause from this rule which is very simple and tells us that we need a digit with the value *0*. In the next iteration we search the grammar for the rules which encode a digit. There are 10 rules — one for each decimal digit. We select the first rule and evaluate the *means* clause, finding that the values match. Thus we generate the terminal 0. The entire parse tree for this example is presented in Figure 6.6.

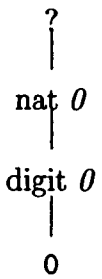


Figure 6.6: Reverse Parse Tree for Value *0*

It is important to distinguish 0 and *0* as they are different. The term *0* denotes a value

of sort `Nat`. The `0` is the terminal from our grammar and is a specific representation of the value 0. This is precisely what makes this approach so valuable as the representation is defined by the user.

Of course we could have used other rules during the generation of the reverse parse tree. Let us explore the possibilities by selecting the second rule during the last step. Then we would evaluate the means clause and realise that *0* is not equal to *s(s(0))*. Then *hep*TEST backtracks and uses another rule. Eventually it would generate the parse tree presented in Figure 6.6.

6.1.6 Test Setup Generation for Tax Example

The parse tree shown in Figure 6.6 is then set into a context. For this system we want create test scripts of this form:

```
Test -- begin
Start tax.exe
Enter ...
```

We expand our syntactic homomorphism and create a grammar rule like in Example 6.7. We use the symbol `nl` to get the desired formatting with newlines.

```
1    start: [Test -- begin nl Start tax.exe nl Enter] nat [nl]
2     means start=calcIncomeTax(nat);
```

Example 6.7: Extension for Syntactic Homomorphism for Tax Example

Using this homomorphism we can continue to create the test script using the reverse parsing technology. The goal is to find a tree which has the `start` symbol as its root and a branch for nat with the value *0*. This is achieved in a single step for this particular example. The complete parse tree is shown in Figure 6.7.

Having created the actual representation for the inputs of the test, we need to do the same for the expected outcome.

6.1.7 Test Outcome Transformation for Tax Example

A similar process to the one described above is applied to the expected outcome. We know the outcome is the result of applying the operation `calcIncomeTax` to the input. We can

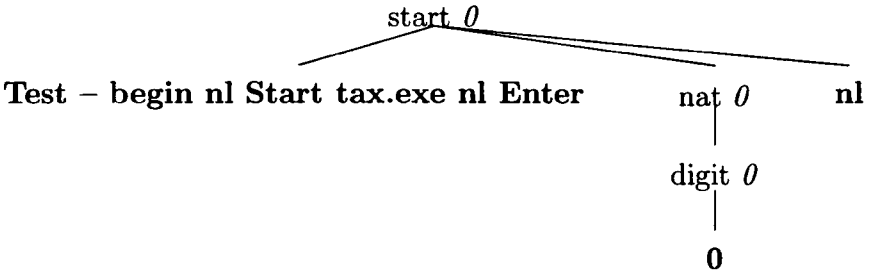


Figure 6.7: Reverse Parse Tree for Entire Test Setup

use LoFT to calculate the expected outcome. For the current example the outcome is 0. Again we need to transform the outcome in a similar way as the input this time for the purpose of test result validation. Here we made the decision to for the outcome the same representation. We can use the same homomorphism as in Example 6.6. But we will use a different completion grammar to distinguish the outcome in the script. Following the same technique we generate the parse tree with the root node outcome as shown in Figure 6.8.

```
outcome: [Expected outcome: ] nat [nl Test -- end]
 means outcome=nat;
```

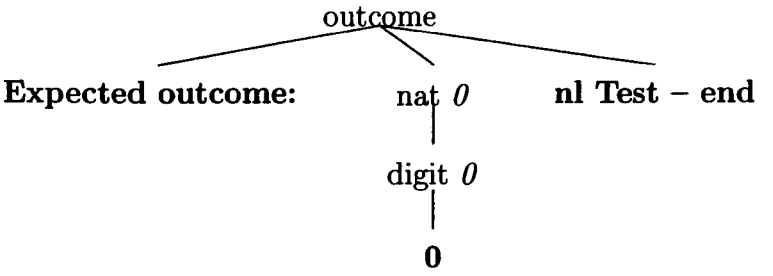


Figure 6.8: Reverse Parse Tree for Expected Outcome

The concrete test is then the sequence of leaves of the reverse parse trees.

If we interleave the generation of test inputs and outcomes then our example would generate the following test:

```
Test -- begin
Start tax.exe
Enter 0
Expected outcome: 0
Test -- end
```

It is noticeable that the details of the exact structure of the test script are all specified in the homomorphisms. In order to accommodate a different language or automated testing we only need to make changes in the definition of the homomorphisms. The well understood way of writing and changing of context-free grammars should make it relatively easy for a user to achieve the desired results.

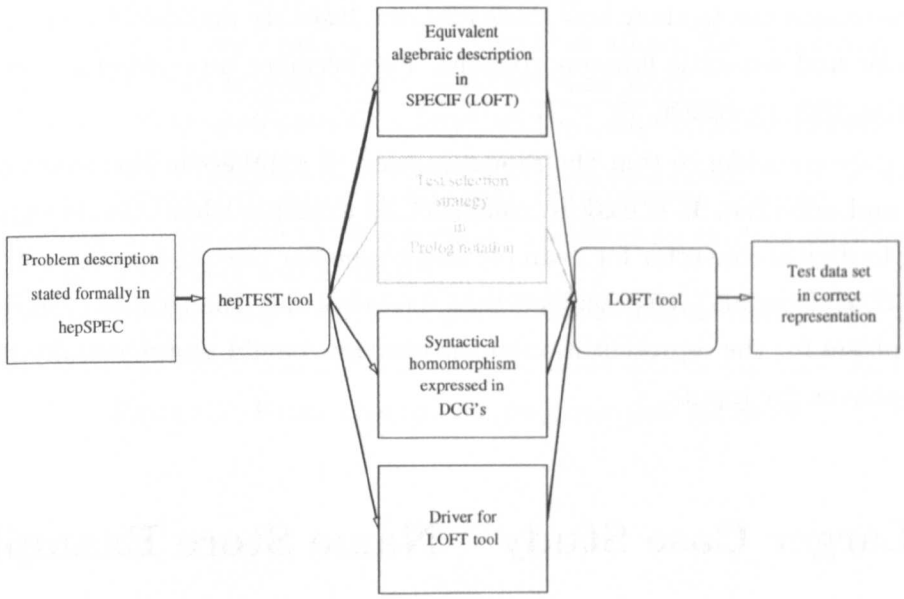


Figure 6.9: Test Automation Setup with *hep*TEST

Figure 6.9 shows the final aspect of our setup, where a driver file is constructed and loaded in LoFT. This driver file contains all the information and steps discussed earlier and the result is a set of tests for our system.

6.1.8 Evaluation of the Tax Study

The example is very simple. However we were able to demonstrate the application of each step of the test generation process.

Starting with the initial step of specifying the system we showed that the specification can be written in a modular way. We can identify components for reuse and therefore ease the creation of specifications in the future. However some common approaches to specifying data types such as natural numbers can lead to difficulties in real world specifications. We saw that writing large numbers with using just `s` and `0` can be very tedious indeed. Later we will have to think of a way to overcome this difficulty for our industrial case study.

We were able to identify the same test cases as Beizer using the same strategy. The advantage was that we could do it automatically. We did not need to specify an order for natural numbers. The partial order proposed in Chapter 5.2 has yield the desired results. An advantage over Beizer's approach has to be the possibility to create the expected outcome automatically.

Then we continued to show how these abstract tests are matched to concrete tests using the syntactic and semantic homomorphisms. Furthermore we created an entire test setup by extending this approach.

It is a pure coincidence that the representation of numbers in this example is the same for input and outcome. It is easy to construct an example where this is untrue. Consider an analog to digital converter for example and remember that its functionality is to convert analog to digital signals. It is assumed that the value only changes its representation. Our homomorphism for the representation of the outcome would then naturally differ from the homomorphisms for inputs.

6.2 Larger Case Study – Name Store Example

We have explained the principles of our approach using a small example in the previous section. Now we set out to verify the approach using a larger case study supplied to us by Dick (Dick, 2001). This case study will show the details of the steps within *hep*TEST. We go on to report on the effectiveness of the technique in a commercial setting in an industrial case study in Section 6.3.

6.2.1 Description of Store Example

The example is a name store for checking membership of a private club. A clerk checks names for membership before allowing access to the club. Names have to be entered and deleted from the store. The name store content is saved from day to day and can be loaded again. Table 6.1 contains the functional system requirements for our example.

The system under test is started by running `run.exe`. The text based menu in Figure 6.10 is provided as a user interface. The user can make entries interactively. Possible commands are `p`, `i`, `d`, `c`, `q`, `l`, and `s`. Some commands require an argument which is requested by the system and supplied by the user. The command `q` terminates the session.

No.	Description
1	The system shall provide a function that allows the Membership Clerk to check whether a given name exists in the name store.
2	The system shall provide a function that allows the Membership Clerk to enter a new name into the name store.
3	The system shall provide a function that allows the Membership Clerk to remove an existing name from the name store.
4	The system shall provide a function that allows the Membership Clerk to list all the names in the name store.
5	The system shall retain membership data between sessions.
6	If the Membership Clerk attempts to register an existing member of the club, a warning will be issued.
7	If the Membership Clerk attempts to remove a name that is not a member of the club, a warning will be issued.

Table 6.1: Functional system requirements for Store

```
==========
| MENU    |
================================================
| Input   |          Function                    |
------------------------------------------------
|   p               PRINTS members names         |
|   i               INSERT member into store     |
|   d               DELETE member from store     |
|   c               CHECKS if member is in store |
|   q               QUITS program                |
|   l               LOADS data from file         |
|   s               SAVES data to file           |
================================================


**
```

Figure 6.10: Menu in System under Test

6.2.2 Specification of Store Example in *hep*SPEC

The first step is to create a specification which describes accurately the desired properties of the system under test.

We developed the previous specification starting from the basic types in a bottom-up manner. Sometimes it is easier to develop a specification from top. We find the basic functional system requirements in Table 6.1. Starting with Requirements R1 - R5 we create the specification `Store` in Example 6.8 and enhance it with more details later on.

```
1   Store is ...
2    sorts Store
3    oprn
4     enterInStore(Store, Member) → Store        -- R2
5     removeFromStore(Store, Member) → Store     -- R3
6     inStore(Member, Store) → Bool              -- R1
7     listMembers(Store) → List                  -- R4
8     loadFromFile(File) → Store                 -- R5
9     saveToFile(Store) → File                   -- R5
10   axioms ...
11  end Store
```

Example 6.8: Top Specification of Store

Example 6.8 shows the initial step of the specification which we will refine to create a complete *hep*SPEC-specification of the system. In Line 2 we introduce the sort `Store` which will represent a name store. Then we list the operators and signatures of the functions we need according to the system requirements from Table 6.1. For example the operator `inStore` is the function requested by requirement number 1. It will check if a name is present in the name store. In the signatures of the operators we use other sorts like `Member`, `Bool`, or `List` which we have not defined yet. We will use our standard specification for `Bool` with the operators `true` and `false`. The system requirements do not contain any specifics regarding names. Normally we would need to clarify a problem like this with a client. For simplicity we chose to use a fixed number of constants as names for members as illustrated in Example 6.9. Surely there is an upper limit on the number of member names which can be entered into this store. The requirements do not provide any information about this restriction. So we had to do some experiments with the system under test to revealed this number to be 10. To be able to test if the system forbids us to enter more than 10 names into a store we need 11 constants. The way how and why these constants are linked to concrete names is discussed as part of test reification for this example in

```
1    Member is Bool with generation
2     sorts Member
3      oprn name1 → Member
4           name2 → Member
5           name3 → Member
6           ...
7           name11 → Member
8     with definition
9      oprn eqMember(Name,Name) -> Bool
10     axioms a,b: Member
11      eqMember(a,a) = true
12      if eqMember(a,b) = false
13       then eqMember(b,a) = false
14
15      eqMember(name1,name2) = false
16      eqMember(name1,name3) = false
17      ...
18   end Member
```

Example 6.9: Simple Specification for Member Names

Section 6.2.5.

We also have to restrict the domains of operations. There is for example an upper limit of 10 on the number of members the name store can hold as mentioned above. This needs to be reflected in a domain condition for the operation of entering a new name into the store. Example 6.10 illustrates restricted operations from our specification. The domain

```
1  with generation
2   oprn enterInStore(x:Store,m:Member
3          iff inStore(m,x)=false
4              lt(numberOfMembers(x),maxMembersInThisStore) = true) → Store
5
6  with definition
7   oprn removeFromStore(x:Store,m:Member
8          iff inStore(m,x)=true) → Store
9        loadFromFile(f:File
10         iff fileExists(f)=true) → Store
```

Example 6.10: Restricted Operations in Store

of `enterInStore` has been restricted in such a way that a member can only be entered if the name is new, that means that it is not already present in the store, and if there is

enough room to store a new name (see Lines 3–4). We make use of auxilliary operators `lt` and `numberOfMembers` to define these properties. This satisfies the Requirement R6 indirectly. We expect our system to issue a warning if the domain condition is not satisfied (see Section 6.2.3 and 6.2.8). The load-from-file operation can only be successful if the file exists. Therefore a restriction is put on the domain of the `loadFromFile` operator. We also do not permit deletion of a member from a store if the name is not already present. This is reflected in the domain condition for `removeFromStore` on Line 8. This restriction is not obvious. The following axiom means that the store does not change if an attempt is made to remove a name that is not present from the store.

$$\texttt{if inStore(m,x)=false then removeFromStore(x,m)} = \texttt{x}$$

But it also means that we would allow this operation and this is contradictory to what the system description (Requirement R7) in Table 6.1 requires from the store.

We need to provide the semantics for the operations we declared. Here we encounter difficulties. The difficulty stems from the usage of LoFT as the equation solver. For example the result of entering names in the store should be independent of the order of operations. The following axiom describes this property:

```
if premises ...
then enterInStore(enterInStore(x,m),n) = enterInStore(enterInStore(x,n),m)
```

There are more axioms which expose this problem. In Example 6.10 we put a restriction on the operator `enterInStore` in Line 3. The following axiom would have been needed to describe the semantics of the operation correctly if we had not restricted the operation.

$$\texttt{enterInStore(enterInStore(x,m),m)} = \texttt{enterInStore(x,m)}$$

The axiom describes the fact that multiple enter operations do not result in multiple entries of the same name in the name store.

The difficulty is that LoFT does not permit axioms between generators (see Section 4.5). However `enterInStore` is a generator for our sort `Store`. To overcome this difficulty we could modify our specification. We could introduce an auxiliary sort, say `Sequence`, to model stores and then use defined operations to model the functions from the system requirements. Equations between defined operations are permitted and we could use the

axiom from above. This resolves the issue but the specification becomes cluttered. We would have to accept this inconvenience if we want to continue to use LoFT.

However we can also use this restriction imposed by LoFT to our advantage. If we do not specify the order-independence property then tests will be generated which should expose this property. Rather than using equivalence classes based on equations between generators as representatives all tests remain relevant. That means, we would test if the order of inserts in the name store does not matter as required. This is our chosen approach in this case. The complete specification can be found in Appendix B.

6.2.3 Test Data Generation for Store Example

The operations Enter, Delete, List, and Check are the core operations of our system. These operations have been mentioned explicitly in the system requirements. We have to test the implementation of these operations. The arguments to these operations are pairs `(Store, Member)`. Our model contains 11 constant members. Elements of sort `Store` are created by the generators `emptyStore` and `enterInStore`.

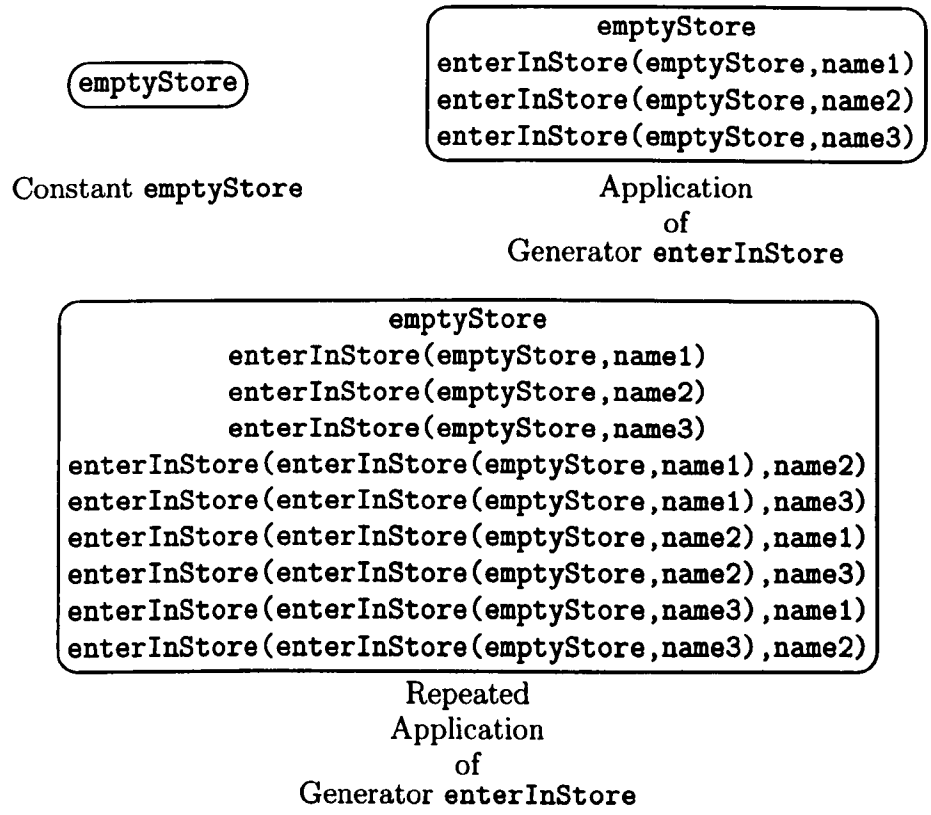


Figure 6.11: Generation of Test Data for Sort `Store`

Figure 6.11 exemplifies, using 3 member names, how the elements of sort `Store` are generated starting with the generator `emptyStore` and repeated application of `enterInStore`. These elements are the basis for test selection in the next step.

6.2.4 Test Data Selection for Store Example

We will construct the partial order for elements of sort `Store` which we generated in Figure 6.11. Figure 6.12 illustrates the partial order resulting from the immediate sub-term relation over elements of sort `Store`. The arrows are pointing to the immediate sub-term element.

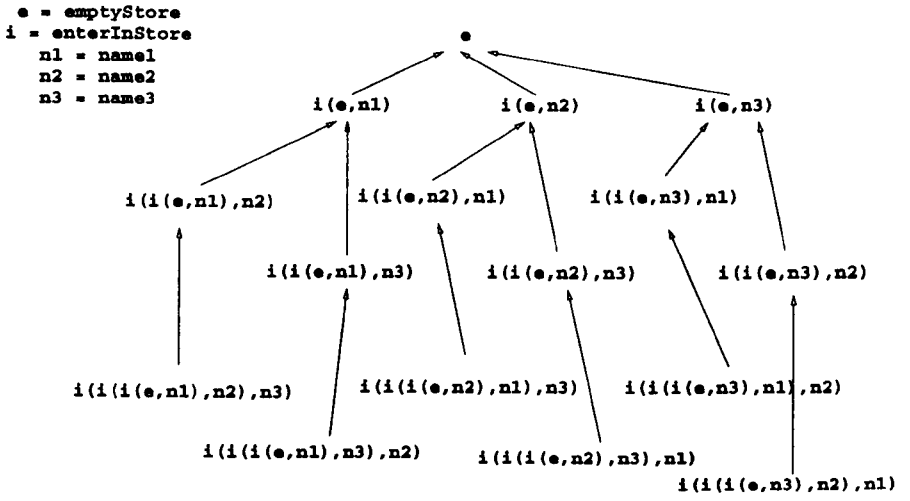


Figure 6.12: Partial Order for Sort `Store` with 3 Member Constants

It is worth noting that this order is already the order on test inputs of sort `Store`. In contrast to Figure 5.6 in Section 5.2.1, we cannot add relations between elements. According to the partial order over elements of sort `Member` all elements are incomparable because there is no element of sort `Member` which is an immediate sub-term of another element of sort `Member`. If we had chosen to model member names in a different way we could have generated a more complex order over elements of sort `Member` and therefore a complex order over test inputs of sort `Store`. For example we could have chosen to use constants for individual characters and to build elements of sort `Member` through concatenation. This choice would have lead to a combinatorial explosion of test cases. To avoid this problem we decided not to include the specifics of names in our theory. The example illustrates that the way the system under test is modelled has an impact on the test data generated.

The operations for entering and deleting member names take two arguments, a store and a member name. According to our theory we need to construct a partial order over pairs (`Set`, `Member`). Figure 6.13 illustrates parts of the resulting order. It becomes a

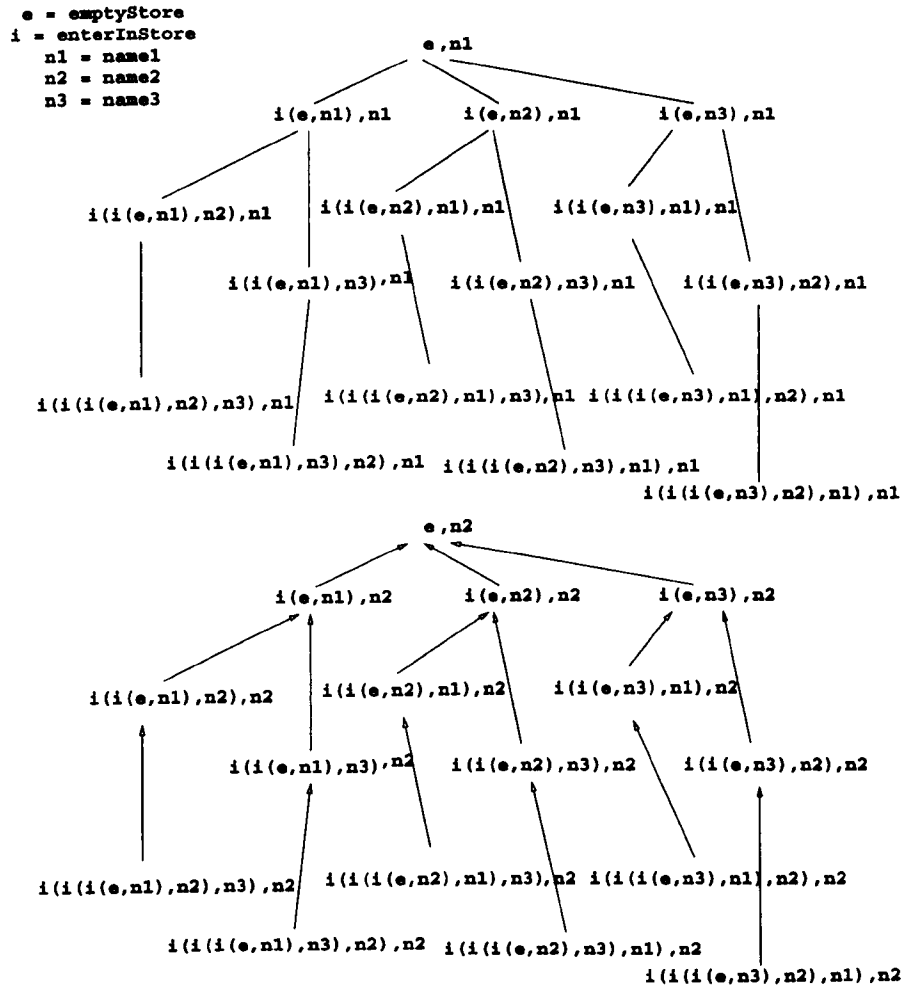


Figure 6.13: Partial Order for Pairs of `Store` and `Member`

forest where each element from `Member` is paired with the tree from Figure 6.12. Visually we could describe it as shadows of the tree from Figure 6.12 which is repeated for each element from the set of `Members`.

Axioms separate this forest into regions which we will use for the purpose of boundary testing. Let us consider Example 6.11 to illustrate this effect for the `removeFromStore` operation. The first axiom selects all elements where we remove the last entered member name (see Line 1–3). In Figure 6.14 we have marked these elements with a box. The second axiom contains those elements where the element to be removed was entered sometime

```
1     if inStore(m,x) = false
2        lt(numberOfMembers(x),maxMembersInThisStore) = true
3      then removeFromStore(enterInStore(x,m),m) = x
4
5     if inStore(m,x) = true
6        lt(numberOfMembers(x),maxMembersInThisStore) = true
7        inStore(n,x) = false
8        eqMember(m,n) = false
9      then removeFromStore(enterInStore(x,n),m)
10           = enterInStore(removeFromStore(x,m),n)
```

Example 6.11: Axioms for `removeFromStore`

before the last one. The grey marked pairs in Figure 6.14 are not part of the domain of

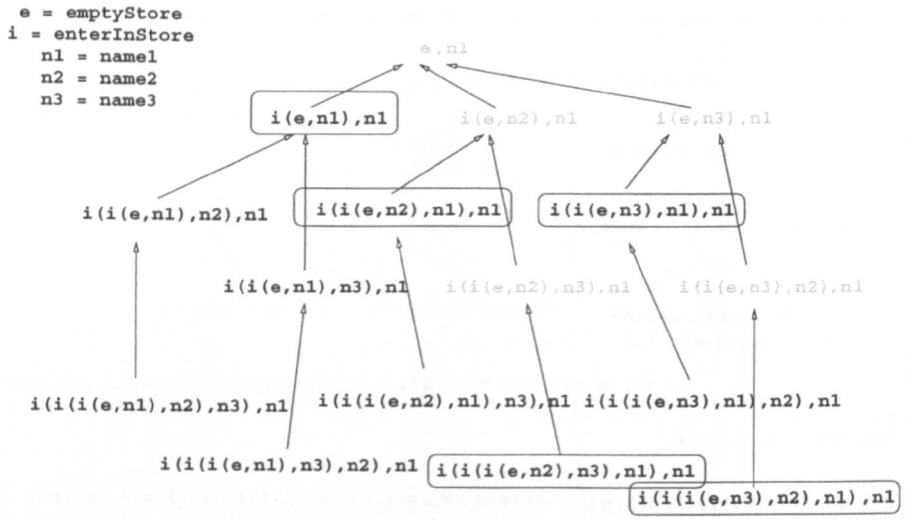


Figure 6.14: Domain split for `removeFromStore(x,name1)`

the `removeFromStore` operator because they are not part of the solution of the domain condition.

From the remaining two valid domain regions we select test cases for positive testing. The Min-Max strategy as presented in Section 5.2.3 would select all test cases from Figure 6.14 from the first domain split because these elements are isolated in the partial order. That means that they are incomparable. From the second domain the remaining elements in Figure 6.14 are selected.

If we choose to test outside of the restricted domain of `removeFromStore`, then the elements neighboring our tests would be selected. From those elements marked grey in Figure 6.14 we would select all because each one is an immediate sub-term of a selected

positive test.

Using the term rewriting system, mainly based upon LoFT, we can compute the outcome for each test. We expect that values from outside the domain will generate an error message. If no error message is generated then we will treat this as a bug.

6.2.5 Test Transformation for Store Example

Up to now we have selected only abstract test cases. We need to create concrete tests for the implementation under test. To show how tests are transformed, consider the following abstract test case from the previous section

```
removeFromStore(enterInStore(enterInStore(emptyStore,name1),name2),name1)
                                        = enterInStore(emptyStore,name2)
```

This test cannot be applied to the system under test directly. It does not fit the text-based menu we have seen in Figure 6.10. The following concrete test input is possible, assuming that a store, containing 9, is present at runtime.

```
d [Newline]
9 [Newline]
```

We decided that the abstract member name **name1** will be represented by the digit 9. From the description of the system we know that the remove operation is executed by selecting d from the text menu. Each input is terminated by a newline character.

First we need to define a syntactic homomorphism that denotes a mapping from the abstract to the concrete test input. The extract from a grammar in Example 6.14 will

```
1    CommandSeq : CommandSeq [d] [nl] Name [nl]
2      where inStore(Name,CommandSeq1) = true
3      means CommandSeq0 = removeFromStore(CommandSeq1 , Name);
4
5    ...
6    Name : [9]
7      means Name = name1;
8    ...
```

Example 6.12: Syntactic Homomorphism for Abstract Tests

achieve the necessary transformation in the following way. Starting with the outermost

term `removeFromStore` we select the grammar rule in Line 1 for `CommandSeq`. Then we solve the equation based on the means clause in Line 3:

```
removeFromStore(enterInStore(enterInStore(emptyStore,name1),name2),name1)
                                        = removeFromStore(CommandSeq1, Name)
```

Equation solving means using the term-rewrite engine based on LoFT. The result of this operation is that the equation holds if

```
CommandSeq1 = enterInStore(enterInStore(emptyStore,name1),name2)
      Name = name1
```

We check that this solution also satisfies the where-clause. In this case it does because `name1` is present in the store. Now we can generate the parse node according to the body of the rule. However two nonterminals remain in this generated tree. One is `Name` and the abstract term for it is `name1`. The grammar rule for `Name` in Line 6 is applicable. The means-clause is solved again with the help of LoFT. In this case the solution is trivial and we do not have to check any where-clause. We can generate the node. If we would have encountered alternatives, then we would have proceeded as described in Section 5.4.3.

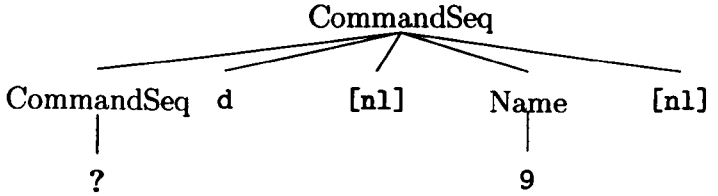


Figure 6.15: Reverse Parse Tree Generation for Concrete Test Input

Figure 6.15 shows the reverse parse tree for this test input. If we consider the sequence of terminals generated so far then we see that this is `d [nl] 9 [nl]` as we expected.

We have not explained yet why `name1` is mapped to the digit `9`. There is no reason. We had to choose 11 concrete representations for the 11 member names in our specification. Initially we were tempted to use common names like "Smith" or "Miller". However the requirements contain no information about how names look like. So a good idea was to generate 11 random strings of characters.

Example 6.13 illustrates another technique we used for names. The system should not distinguish names which were obviously mistyped. We consider all of the names in

```
1     Name : [Smith]
2       means Name = name3;
3
4     Name : [ Smith]
5       means Name = name3;
6
7     Name : [Smith ]
8       means Name = name3;
9
10    Name : [smith]
11      means Name = name3;
12
13    Name : [SMITH]
14      means Name = name3;
15    ...
```

Example 6.13: Mapping variants to a single abstract term

Example 6.13 to be the abstract name **name3**. Two questions remain. First how does the *hep*TEST tool cope with this mapping? The rules are chosen randomly until we find a solution. Therefore **name3** will be mapped to different names at different places in the test. Second how does this effect our theory. We could argue that this mapping is not a homomorphism anymore. These rules seems to be a one-to-many mapping and therefore not a homomorphism. However this issue is resolved by arguing that it is a mapping from a value to a set of names, which are all equivalent, making it a homomorphism again.

We have calculated the expected outcome after test case generation and selection steps. However we cannot assume that whenever the test sequence from Figure 6.15 is executed the expected outcome is produced. We have to ensure that the abstract argument for store is actually present at runtime. This is achieved during test setup generation where the remaining leaf from the tree in Figure 6.15 is substituted by a branch.

6.2.6 Test Setup Generation for Store Example

In the previous section we transformed the abstract test

```
removeFromStore(enterInStore(enterInStore(emptyStore,name1),name2),name1)
                                        = enterInStore(emptyStore,name2)
```

into the concrete test

```
d [Newline]
9 [Newline]
```

and noted that we would have to ensure that the concrete form of the abstract term

```
enterInStore(enterInStore(emptyStore,name1),name2)
```

is present at runtime. Figure 6.15 shows the reverse parse tree that was created during test input transformation. The nonterminal representing the setup state has not yet been substituted by a tree.

```
1   Sentence : CommandSeq [s] [nl] [q] nl]
2    means Sentence = CommandSeq;
3
4
5   CommandSeq : [s] [nl]
6     means ComandSeq = emptyStore;
7
8   CommandSeq : CommandSeq [i] [nl] Name [nl]
9     where inStore(Name,CommandSeq1) = false
10          lt(numberOfMembers(CommandSeq1),maxMembersInThisStore) = true
11    means CommandSeq0 = enterInStore(CommandSeq1 ,Name);
12
13  CommandSeq : CommandSeq [d] [nl] Name [nl]
14    where inStore(Name,CommandSeq1) = true
15    means CommandSeq0 = removeFromStore(CommandSeq1 ,Name);
16  ...
17
18  Name : [9]
19    means Name = name1;
20
21  Name : [$DF%x]
22    means Name = name2;
23  ...
```

Example 6.14: Parts of the Syntactic Homomorphism for Store Example

Example 6.14 shows part of the grammar we use to create the setup. The grammar is an extension to the grammar in Example 6.12 which we use to convert the abstract tests into concrete tests. The rule in Lines 8 – 11 links the operator `enterInStore` to the concrete command `i` and the where clause ensures that we cannot create concrete name stores which have no equivalent in the abstract world.

To complete the test setup we have to substitute the nonterminal CommandSeq by a tree. The same principles are applied as in the test transformation step. The node is labeled which the abstract term

```
enterInStore(enterInStore(emptyStore,name1),name2).
```

We begin with the outermost term **enterInStore** and select the rule from Line 8 from the grammar in Example 6.14. We have to solve the following equation:

```
enterInStore(enterInStore(emptyStore,name1),name2) =
                                    enterInStore(CommandSeq1 ,Name)
```

A solution to this equation is given by:

```
CommandSeq1 = enterInStore(emptyStore,name1)
       Name = name2
```

This solution also satisfies the where-clause. We can create the tree node and have to continue the process for the remaining nonterminals. The rule for **Name** in Lines 24–25 in Example 6.14 will create the node labelled with **name2**. And the nonterminal **CommandSeq** is substituted by a tree similar to the process described. At the end we need to create a tree for the node labelled **emptySet**. The rule in Lines 5–6 from Example 6.14 creates the command sequence `s [nl]`. It might be confusing why this is necessary. This initial save-to-file operation creates an empty store on the hard drive. The existence of an empty store is a workaround for a bug in the system under test. The program crashes if no file is present when a load operation is carried out. Other bugs found in the system are discussed in Section 6.2.8.

The complete tree for the test setup is presented in Figure 6.16. The following sequence of terminals is created by a walk through the tree and constitutes the concrete test input.

```
s  i  9  i  $DF%x  d  9  s  q
```

This tree is completed by the command sequence `s q` generated by the rule in Lines 1–2 from Example 6.15 to terminate the test. We add a save operation to have a persistent contents of the store which otherwise would only exist in the memory of the computer. The `q` is necessary to avoid a bug which prevents us from executing tests automatically

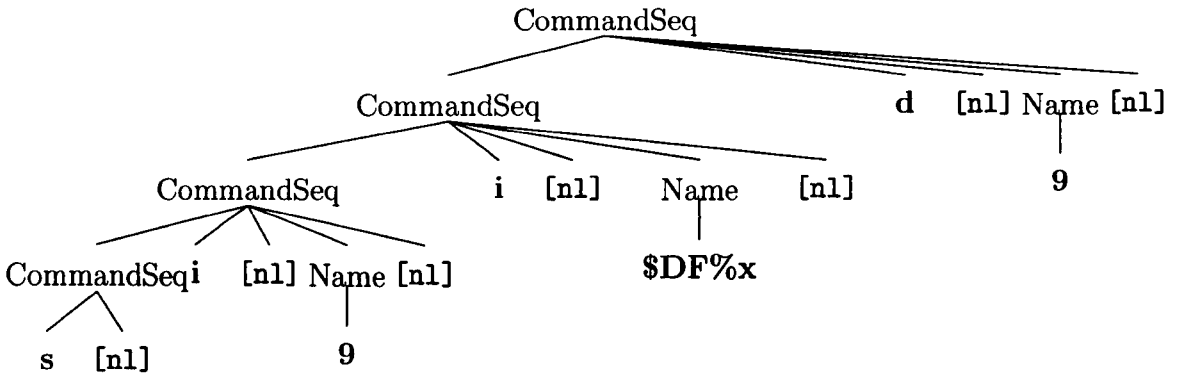


Figure 6.16: Test Setup for Store Example

(Section 6.2.8). We choose to save the store and terminate each test sequence to be able to execute them automatically in batch mode.

6.2.7 Test Outcome Transformation for Store Example

In the previous section we transformed the left-hand side of the abstract test

```
removeFromStore(enterInStore(enterInStore(emptyStore,name1),name2),name1)
                                                    = enterInStore(emptyStore,name2)
```

into a concrete test input with test setup. Now we need to transform the right-hand side of the equation to be able to compare the outcome in the concrete world. Example 6.15

```
1    Result :
2      means Result = emptyStore;
3
4    Result : Result Name [nl]
5      where inStore(Name,Result1) = false
6            lt(numberOfMembers(Result1),maxMembersInThisStore) = true
7      means Result0 = enterInStore(Result1,Name);
8    ...
```

Example 6.15: Semantic Homomorphism for Sort Store

contains part of the grammar that contains the outcome tranformation. The tranformation process itself does not differ from the transformation of the input. The names in the store have the same concrete representation as in the input domain. We used the same nonterminal to denote this fact and use part of the grammar for the input transformation

to transform names in the semantic homomorphism. This is for convenience and to improve maintainability of the homomorphism. If we change the representation for `name1` then this change is present in both homomorphisms. The operator `enterInStore` is transformed differently from the syntactic homomorphism. In the semantic homomorphism it produces a plain list of names separated by newlines. In the syntactic homomorphism the generation of a store was done by the command `i`. Thus it is even in this simple example necessary to distinguish the input and output transformations.

```
1    Bool: [YES, name is in store]
2      means Bool = true;
3    Bool: [NO, name is not in store]
4      means Bool = false;
5    ...
```

Example 6.16: Semantic Homomorphism for Sort Bool

We also have operations which do not have `Store` as their range. The operator `inStore` returns a Boolean value. Example 6.16 shows how such a value can be transformed. The concrete representation is a message string on the screen.

The transformation of test outcome was not trivial in this case. The difficulties stem from the fact that the system was not designed with automated testing in mind. The operations enter, remove, and save modify the content of the inner state of the system and it is available in a persistent file form only at the end of the test sequence. The result of other operations like `inStore` can only be monitored on screen. So we decided to redirect the output to a file and to use the shell command `diff` to evaluate the result. Unfortunately we have now two actual outcomes: the data base file and the screen dump. However `diff` does not know which operation will modify which contents. So we generated both a screen entry and the content of the store file in the expected outcome. A thorough discussion of the test results can be found in the next section.

6.2.8 Test Execution and Result Validation for Store Example

Our test environment was set up in the following way. Each of the tests was stored in a separate file. The expected outcome for each test was stored in a file also. The file number related the test input to the expected outcome. The expected result for tstfile.00001 was stored in outcome.00001 for example. This function was integrated into the *hep*TEST system.

The tests were run automatically in batch mode. At the start of every run of the system the internal store is empty. The outcome of each test was stored in a separate file. Before a `diff` script was used to compare actual and expected outcome the files where sorted by `sort`. This is necessary because we do not know in which order the member names will be stored in the file. A line documenting the result of the test evaluation is added to a report.

The following errors were detected during the automated test runs.

Bug: ***Loading membership data base.*** A crash occurs if there is no membership data base present. We mentioned this bug and our workaround in Section 6.2.6. The test was a negative test which violated the domain condition of `loadFromFile`.

Test input

```
l
s
q
```

Expected outcome

```
ERROR: *
```

Actual outcome The program crashed and creates a dump.

We used the expected outcome `ERROR: *` to mark the result of a negative test.

Bug: ***Program does not terminate on EOF (end of file).***
Executing the test cases in batch mode required the addition of the quit command to each test script. This was a positive test for the axiom in Example 6.11 on Lines 1–3.

Test input

```
s
i
Smith
d
Smith
s
```

Expected outcome An empty output_data file.

Actual outcome Program loops infinitely.

These initial difficulties were overcome by modifying the grammar as discussed in Sections 6.2.6 and 6.2.7.

Bug: ***Empty strings are falsely recognised as names.*** The system reports falsely that a name of length 0 is already present in the store. When we generated the random names this was one of the values created. The requirements do not mention this case.

Test input

```
s
i
Smith
i
[newline]
d
[newline]
s
q
```

Expected outcome

```
Smith
```

Actual outcome

```
ERROR: Entry already exists in store
```

Bug: ***Inserting, checking, deleting names with special characters.*** Member names with German umlaut and other characters such as ‘ and the paragraph sign crash the system.

Test input

```
s
i
Ö
d
Ö
```

```
s
q
```

Expected outcome An empty output_data file.

Actual outcome System crashed and creates a dump.

Quite a number of tests with special characters were executed. Some, like inserting `Müller` (a common German surname) worked while others failed.

Bug: ***Member names are case-sensitive.*** We allowed for mistyped names in our syntactic grammar. The system distinguished them. The test was created because the representatives for `name3` were chosen randomly as discussed in Section 6.2.6.

Test input

```
s
i Smith
d SMITH
s
q
```

Expected outcome An empty output_data file.

Actual outcome ERROR: Entry not in store

Bug: ***Inserting members with long names.*** The system cannot handle long member names.

Test input

```
s
i
VeryLongNameWithCharacters0123456789012345678
s
q
```

Expected outcome

```
VeryLongNameWithCharacters0123456789012345678
```

Actual outcome The program crashes and creates a dump.

Bug: ***Member name not found in store.*** This bug causes the system not to find a name which is obviously entered. This test case is a positive border test of the `removeFromStore` operation.

Test input

```
s
i
b1ARZfJ
i
%5
i
)Q
i
sR
i
4jf"
i
&yBö
i
dBÖ
i
1v
i
JK%p9
i
9
d
9
s
q
```

Expected outcome

```
b1ARZfJ
%5
)Q
```

```
sR
4jf"
&yBö
dBÖ
1v
JK%p9
```

Actual outcome ERROR: Entry not in store

Bug: ***System crashes during save operation.***

Test input

```
s
i
aRwIPeD
i
S58OmLM
i
cwQ
i
4swu
i
CD41Y
i
awu
i
Rq
i
4U
i
Uhdl
i
7H
d
7P1qfXcgLsbqf
s
```

Expected outcome

```
aRwIPeD
S580mLM
cwQ
4swu
CD41Y
awu
Rq
4U
Uhdl
7H
```

Actual outcome System crashes and creates a dump.

We never suspected that this could happen. We relied in our test execution setup that the save operation would always produce a correct image of the store contents.

All of the tests are reproduceable and can be used for debugging purposes. We can see that even such a program of moderate complexity can create quite a number of errors.

6.2.9 Evaluation of Store Case Study

We are happy to have received this example. It provided a real challenge where we were able to demonstrate the usefulness of our approach. On the other hand we gained a deeper understanding of the limitations of this particular approach.

First, training is required in the *hep*SPEC formalism like for any other approach based on a formal system. The creation of a formal specification is an additional step compared to non-formal approaches. A certain amount of "algebraic" thinking is required to get the specification right.

Second, *hep*TEST is unable to detect errors due to real-time constraints or concurrency. It is also difficult to generate tests directed explicitly at exposing specific faults, for example, errors created through memory leakage.

Third, the choice to reify the expected outcome and to use an external validator is an additional source of errors. It might have been better to include the validation into the formal system by using retrieve instead of reification. However as we discussed in Section 3.5 this would mean that all testers need to receive training in the *hep*SPEC approach, that

the test environment has to be installed on-site. This might limit the acceptance of the *hep*TEST system considerably.

The amount of generated test cases using the Min-Max selection strategy cannot be reduced if our partial order is flat as the example of member names showed. All names are incomparable and so the Min-Max testing strategy will not work. We however retain the advantage of automated test data generation and test outcome computation.

A serious difficulty is the assessment of test results. It is not an easy task to recognise that different failures have been caused by a common bug. A good example is the bug with names containing special characters. There might be hundreds or thousands of tests that reveal this error. A human tester could do this classification fairly easy by close inspection of test results in contrast to an automated solution.

Furthermore, for the time being the implementation of *hep*TEST is limited by the choice of LoFT and ECL^{i}PS^e tools needed to compute the tests. Especially the restriction on axioms between generators hampers the creation of elegant specifications. This might however be overcome in the future.

Can we find all bugs in this example by using our approach? Definitely not. Those bugs which are not covered by the specification were not detected. For example a usability feature like save-before-quit has not been specified and therefore not tested. As a matter of fact it has not been implemented in the system under test either.

A final remark: the case study confirmed that design for testability is necessary if automated testing shall become an intrinsic part of the software development process.

We have so far demonstrated the implementation working as expected in the case study. Now we go on to report a larger industrial-sized case study. We expect to face challenges when specifying the system, we need to determine if the partial order will yield "good" test cases for complex data types and see if the homomorphisms are flexible enough for a real world setting.

6.3 The Industrial-sized Case Study

Now we set out to demonstrate the potential for automated test data generation with *hep*TEST using an industrial example. The example we chose has been supplied to us as an industrial case study. It focuses on a software system for setting up cartridges with information from a database. We call this system basic cartridge setup (BCS).

In Section 6.3.1 we indicate what part of the BCS functionality formed the basis of our

study. In Section 6.3.2 we introduce some examples from the specification of our BCS. Then we follow the testing steps and point to success and difficulties experienced during the case study before we summarise the lessons from the industrial case study.

6.3.1 Project Description

The project we used for the case study is part of a larger software system. An organisation like the Coast Guard may have the duty to mount a large search and rescue mission, then they need it to be planned and executed fast, reliable, and organised to achieve the goal of the mission. Many vehicles such as ships of different size, aircrafts, helicopter but also vehicles on land may be involved. They need to know precisely where to be and when. A group of aircrafts may need to scan a marked area and they have to stick to a devised plan otherwise parts of the area may not be covered. Such strategic missions are planned well and the mission is divided into smaller parts and assigned to teams using a mission planning software. Such a mission support system has been created and we looked at a new software part to enhance the capabilities of this mission control system.

The BCS is a system which interfaces between the Modular Mission Support System (MMSS) and the Data Transfer Cartridge (DTC) via a Data Transfer Module Interface (DTMI). The data is transfered from the MMSS where the assigned missions are stored to the DTC to be used in a vehicle during operation. It does allow a fast and reliable access to an assignment and transfers data to a cartridge without the need of manual copying. The important aspect of the BCS is to ensure that data is transfered consistently and that the different units are converted correctly. The difficulty stems from the fact that cartridges are specific for each vehicle type and do not necessarily use the same units as the planning software. Some data present in the planning software cannot be stored into a cartridge and we need to ensure that this is obeyed. Figure 6.17 presents an overview of the system setup.

There is a variety of DTC's for different vehicle types. The design team made the decision to implement a BCS for each DTC as only one vehicle type is likely to be present at one base. We focussed on one particular implementation of the BCS. The data from the mission planner is stored in a database. The BCS is accessing this data base and retrieving the entries for particular assignments. An assignment or mission has an identifier and contains data about the vehicles involved in that mission. Each vehicle has an identifier, a route containing information where to go, information about communication frequencies and other important data. The BCS we looked at was able to store waypoints for the route

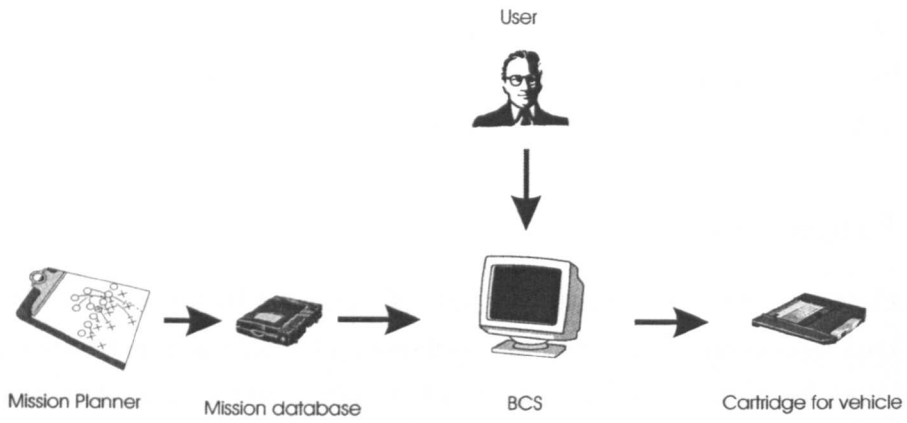


Figure 6.17: Overview of Base Cartridge System (BCS)

information and communication frequencies.

When the user starts the BCS a graphical user interface is displayed (see Figure 6.18). The user can choose to select a mission, or to exit the program. If he chooses to select a mission, a window is displayed with the missions found in the database. The user selects his vehicle from the current mission. Once the selection is completed, the data can be viewed or loaded to the cartridge (DTC). The transfer of data is bidirectional. However, in the interest of brevity, we focused on the uploading of data onto the DTC only.

Parts of the specification of our BCS will be presented in the next section. The complete specification can be found in Appendix C.

6.3.2 Specification of BCS Case Study in *hep*SPEC

This section covers the aspects of building a model of the system under test. We used our framework to produce a formal description of the BCS.

First we identify the main data structures. We obviously will have an object representing the database in MMSS. The database stores items belonging to missions. In each mission there are several vehicles involved. We have for each vehicle a sequence of waypoints representing the route and a set of communication frequencies. These database items are extracted for a specific vehicle by the operator. He can then amend the information manually. A helicopter pilot might wish to add alternative landing sites for emergency landings or possible pick-up places for a ground crew. Then these data items are stored on a DTC.

We will not present each individual data item, but focus on those which yielded inter-

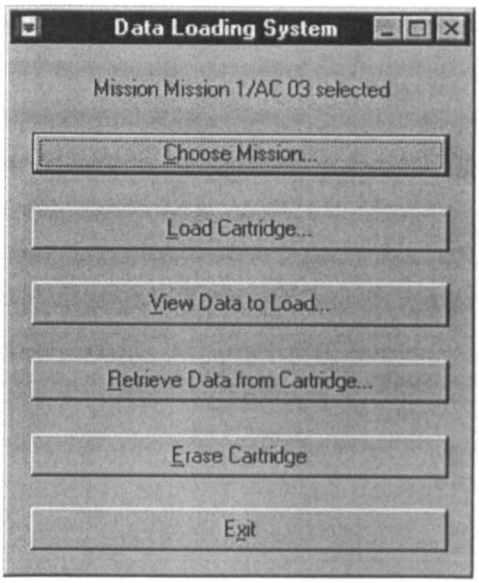


Figure 6.18: Graphical User Interface for BCS

esting results or overcame initial difficulties.

In our previous example in Section 6.1 we instantly ran into difficulties specifying large numbers. Large numbers were needed in this case study too. A waypoint for example has 3 coordinates, longitude, latitude and height. Because our range of vehicles contained also aircrafts heights of waypoints could reach 10000 ft. Even longitude and latitude range up to 180 and 90 degrees respectively. So the first challenge was to specify numbers which would allow us to write even large ones conveniently.

We used our own string arithmetic, very much like the one everybody learns at school. Example 6.17 shows the introduction of 3 new operators to `Nat` which is familiar from the previous example. A constant `10` representing the natural number 10 is specified. We found that using such abbreviations made our specification easier to read and better to maintain. The constant is is followed by a division by 10 operator and a modulo 10 operator. These are also introduced for the same reasons as the operator `10`.

Using these new operators we specified addition for single digits including overflow. Example 6.18 shows the specification. The operator `add_3` takes three arguments, the two operands and the value of the carry. It is possible to write down an entire table for single digit addition with carry, but we thought it was more intuitive this way and reduced the size of the specification.

```
1   Nat is Bool with generation
2    ⋮
3    with definition
4     oprn
5         10 → Nat
6     mod_10(Nat)→ Nat
7     div_10(Nat)→ Nat
8    axioms m:Nat
9     10 = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))))
10    if lt(m,10) = True
11     then mod_10(m) = m
12    if lt(m,10) = False
13     then mod_10(m) = mod_10(sub(m,10))
14    if lt(m,10) = True
15     then div_10(m) = 0
16    if lt(m,10) = False
17     then div_10(m) = succ(div_10(sub(m,10)))
18  end Nat
```

Example 6.17: Amending hep-specification Nat

```
1   NatPair is Nat with generation
2    sorts TNatPair
3    oprn mkPair(Nat, Nat) → TNatPair
4    with definition
5     oprn carry(TNatPair) → Nat
6          result(TNatPair) → Nat
7    axioms m,n:Nat
8      carry(mkPair(m,n)) = m
9     result(mkPair(m,n)) = n
10   with definition
11    oprn add_3(Nat, Nat, Nat) → TNatPair
12   axioms m,n,r,c:Nat
13    if r = add(add(m,n),c)
14     then add_3(m,n,c) = mkPair(div_10(r), mod_10(r))
15  end NatPair
```

Example 6.18: Specifying Single Digit Addition with Carry

Then we introduced DigitSeq and defined the arithmetic operations for equally long sequences of digits. Equal length makes it very easy to define the operations, but we are forced to extend it to unequally long sequences. Example 6.19 shows parts of the specification. We use a trick which is to normalise the length of digit sequences before

```
1   IntegerArithmetic is DigitSeq with generation
2    sorts TIntNumber
3    oprn
4     int(s1:TDigitSeq iff lead_0(s1) = False) → TIntNumber
5
6    with definition
7    oprn
8     i_add(TIntNumber, TIntNumber) → TIntNumber
9     i_mul(TIntNumber, TIntNumber) → TIntNumber
10    i_leq(TIntNumber, TIntNumber) → Bool
11     i_lt(TIntNumber, TIntNumber) → Bool
12
13   axioms s1,s2:TDigitSeq,i1:TIntNumber
14      if lead_0(s1) = False
15         lead_0(s2) = False
16       then i_add(int(s1),int(s2)) =
17          int(dplus(l_normalize(s1,s_length(s2))
18                   ,l_normalize(s2,s_length(s1)),0))
19    ⋮
```

Example 6.19: String Arithmetic for Integers Based on Sequences of Digits

we apply the operation dplus from DigitSeq. We prepend enough zeros to make the two digit sequences equally long. Now we can use the operations defined in DigitSeq to define operations on TIntNumber.

Through step-wise construction we are able to deal with small enough chucks of information at a time giving us the opportunity to handle them with ease. In Example 6.20 we want to demonstrate how we handle the combination of data types in a new one. The sort Waypoint has only one generator (mkWaypoint) which combines the information of several data items into a single object. Then we define a set of getter-operators, e.g. getWaypointID to retrieve data items from the complex object. We will return to waypoints in the following section to discuss the efficiency of our proposed partial order on data items.

```
1   Waypoint is WaypointID with Longitude with Latitude
2            with Elevation with generation
3     sorts Waypoint
4     oprn mkWaypoint(WaypointID,Longitude,Latitude,Elevation)
5                                                    → Waypoint
6     with definition
7     oprn getWaypointID(Waypoint) → WaypointID
8          getWaypointLongitude(Waypoint) → Longitude
9          getWaypointLatitude(Waypoint) → Latitude
10         getWaypointElevation(Waypoint) → Elevation
11    ⋮
```

Example 6.20: Specification of Waypoints

6.3.3 Test Data Generation for Case Study

Test input generation is more complicated in this example than in the previous one. The difficulty arose from the increased number of generators. In the prevoius example we had only 2 generators. Nevertheless with our tool setup we are able to use the same technology as presented before to generate tests for the BCS system.

To create a sequence of digits we only need to use **sq** as an initial generator and keep appending with the generator **app**. However before using **sq** we need to generate an item of sort **Nat**. How this is done we showed in Section 6.1.2. Here we have to observe the restrictions of the generators. For example if we create an item of sort **TIntNumber** through **int** and the generators of **TDigitSeq** we need to observe the restrictions on **int** (see Example 6.19) as well as those on **sq** and **app**. Following this way we create only valid items for a given sort.

In Chapter 5 we proposed an order on items of the same sort. Using the definition of waypoints in Example 6.20 we want to show how our decision to use that particular order influences the ability to minimise test sets. From the definition of **Waypoint** we see that it has only one generator. If we would have used the order based only on the generator for sort **Waypoint** as we initially proposed in Chapter 5.2 then we would not be able to choose between values of sort **Waypoint**. They would be all unique and in no relation to each other. That in turn means that all of them need to be selected for testing. Using our more elaborate order based on the structure of the data item using the orders of the underlying sorts recursively we are able to sort even waypoint values efficiently. A more general discussion of the testing strategy follows in the next section.

6.3.4 Test Data Selection for Case Study

*hep*TEST supports a number of testing strategies. All of these strategies have advantages and disadvantages and have to be selected to conform to some assumptions about the software.

For example random testing could be applied when no knowledge about the software design and implementation exists. The downside is the number of tests generated. Random generation in *hep*TEST will produce a random sequence of simple entities. The simple entities are derived from the test conversion. Terminals from the grammar are selected and placed in a random order. Such an approach is meaningful for robustness testing where the goal is to break the software. No expected test outcome is generated as the tests can be accessed in a simple manner. If the software is still operational, then the test was successful otherwise the test failed (N.P.Kropp et al., 1998).

Random generation is a very powerful method, but also requires a lot of resources. It becomes infeasible in system testing or when the tests are executed manually or test execution is expensive. To reduce the test set we can apply context-free testing. The term stems from the test data conversion approach as parts of the conversion are described by a context-free grammar. This grammar is utilised by *hep*TEST to generate tests which explore the context conditions of the grammar. Such a strategy could be applied to test a compiler for example (Burgess, 1993; Bauer and Finger, 1979). The context-free grammar of the language will be used to generate valid sentences. Some of the test inputs should be rejected by the compiler. These are the sentences which violate the context conditions of the language. Take for example the very common constraint in programming languages that a variable needs to be declared before it is used. This constraint is usually implemented by using look-up tables. The context-free grammar does not contain this constraint, so tests will be generated which will violate this particular constraint. The advantage of context-free testing is that the number of tests is reduced. It also focuses more on the functionality of the system rather than on its robustness. This technique would be applied when the sequence of operations is under test. The underlying assumption is that some sort of development technique ensures that context free sequences are recognised correctly so that the omitted tests from random generation would have passed anyway.

Context-free testing would still create to many tests for our system. Tests would be included which would not focus at the particularities of the system considered here. The grammar rules would produce tests which voilate the where-clauses of the grammar. However in this case study it is important to obey the restrictions because it does not make

sense to go 370 degree north for example when 90 degrees north is the limit and cannot be exceeded. We want to use boundary analysis to exploit the fact that we have knowledge about the behaviour of the system. Using the axioms which define each operation we can derive domains and sub-domains as presented in Chapter 5.2.2 and shown in the previous example. From these domains we propose to choose test inputs according to a specific strategy. Min-Max values is a common strategy for boundary testing. We discussed it in detail in Chapter 5.2.3.

We could also generate just-out-of-boundary tests following Beizer's proposal using the partial order on sorts (Beizer, 1995). If the following Prolog query is executed with `X` bound to a maximum value then `Y` will compute to an just-out-of-boundary value.

```
?- im_subterm_of_sort_NAT(X,Y).
```

Equally if the query is executed with `Y` bound to a minimum value then `X` will compute to a just-out-of-boundary value should it exist. If `Y` is a constant generator then there are no smaller values. The predicate `im_subterm_of_sort_*` is available for each sort so we are not restricted to `NAT`.

In practice, it is up to the engineer to select the strategy believed to be most appropriate for each situation. This choice is guided by assumptions on the system under test and its environment. For the case study we can assume that the data base (MMSS) is implemented correctly as well as the interface provided to us. The data base has been part of another product and has been used extensively for many years. This should give us enough confidence to omit tests which exercise the data base outside its boundaries (negative tests). We can focus our tests on the transformations within the BCS. Choosing the max-min-average test strategy and passing these tests should give us sufficient confidence in our system.

6.3.5 Test Transformation for Case Study

For now let us assume that we test operation `op` and the test we selected is of the form `op(in) = out`. This would mean that we have to supply the operation with inputs in the correct format and apply a transformation to the outcome for test result validation.

We introduced input and outcome transformations for this purpose in Chapter 5. We want to show now that this technique is applicable in an industrial setting. First we need to make a decision about our concrete tests. For this case study we choose to use manual test execution thus our concrete inputs should be human readable. For the purpose of

testing we were supplied with an module of the cartridge software that would dump into a plain text file. We were able to use ASCII texts as concrete representations of our tests.

Let us look at a concrete example from our case study. Consider for example that we need to generate a value for `mkLongitude(X,Y)` where `X,Y` is fixed. Here is a part of the grammar describing the test conversion.

```
longitude: [longitude: ] [W ] decimal [ deg ]
     where decimal = ddot(d,f) & 0 < d <= 180 & length(f) <= 3
     means longitude = mkLongitude(west,decimal)
  | [longitude: ] [E ] decimal [ deg ]
     where decimal = ddot(d,f) & d <= 180 & length(f) <= 3
     means longitude = mkLongitude(east,decimal);
```

Looking at the rules and the means-part in particular we see that if `X` is `west` we should use the first rule. Then having generated the terminals `longitude: W` we use the value of `Y` to generate a `decimal` before completing the sentence with `deg`. The same process is applied to generate a decimal. The result could be `longitude: W 10.0 deg`. In this way we build a parse tree from top to bottom until all nonterminals are substituted. The entries on the leaves comprise the concrete test input.

A similar transformation is applied to the outcome of a test. We could use the same test convertion but in general data of even the same sort will have a different format in the range domain. Thus we use the so called semantic homomorphism to produce the test outcome in the required format for test result evaluation. In this case the data is translated into a sequence of digits and test validation can be achieved using the Unix `diff` command for example. The data which is presented on the screen will be formated for manual test result evaluation.

Later we could have decided to use automatic test execution tools for regression testing, then the test convertions would be transformed so that test inputs are translated into a sequence of scripting instructions. This way the test execution remains a flexible affair. The conversion of tests into different formats allows for a smooth transition with little effort.

In this case study we could have produced SQL statements to create the mission data base. Unfortunately we did not have access to the internal structure of the data base. Test execution was affected by this too.

6.3.6 Test Execution and Result Validation for Case Study

This case study did execute only a small number of tests.

The main reason for that is that we did not have access to the mission planner software. The *hep*TEST-tool produces test cases in the desired input format and in addition the entire test setup. Thus the test cases produced contain instructions to the user on how to generate the desired mission database. We were limited to the tests which we could perform through the user interface of the BCS. So instead of testing a variety of mission databases we used a constant database and applied the functions of selection and transfer to that. This way we could execute only 10% of our generated test cases. This shows that contractual agreements can influence testing or the other way round. Test planning has to start at the beginning of the project, but this opens a whole new chapter of dicussions and would extend the scope of this thesis by far.

However we were able to identify a number of issues, which have been solved in subsequent software development stages. One of the problems was that longitude and latitude were displayed incorrectly. The orientations north/south and east/west were exchanged. This problem might have been spotted by the testers, but the author himself was in doubt and had to review this subject before he was sure that a problem was identified. A tester under pressure might not have been that thorough. And this is the biggest advantage of having an expected test result present in the same format as the software is producing it.

An other issue was dicovered during conversion of values into different units. Longitude and latitude can be entered using degrees, degrees and minutes, or degrees, minutes and seconds. During conversion an error was discovered that resulted in values where there were 61 seconds in a minute, like in 9°01′60″. However we know from the specification that a minute counts from 0 to 59 seconds only. We were able to discover the fault and provide a correction before the product was released to a customer.

6.3.7 Conclusions

First it is important to note that tool support became crucial in this case study. Only owing to the possibility to animate the specification was it possible to create a description which not only was formal correct but also presented the actual desired properties of the software. The *hep*SPEC syntax checker, *hep*TEST-tool and LoFT all contributed to the success.

Some of the requirements needed to be clarified and properties of data investigated.

These questions contributed to the accuracy of the specification. In general we can claim that the formalisation of the requirements gave rise to these questions which might have been overlooked or required further analysis in later stages of the development process otherwise.

In the area of test generation we were able to demonstrate that *hep*TEST is a suitable approach to generation of system tests. The main improvements over traditional testing are:

- Systematic generation of test cases following a testing strategy. The *hep*TEST approach used the specification to identify the boundaries vital for the chosen test strategy. The tests are generated fully automatic thus reducing the costs of test generation and the testing costs as a whole.

- The generation of an expected outcome eases test result validation. A simple comparison can answer the question of test success or failure.

- The existence of expected outcome makes a fully automated test execution and test result validation possible. This can increase the number of tests applied dramatically and improve so the confidence into the correctness of the system under test.

The flexibility of the test conversions has proven to be adequate for the needs of this project. Also the application of boundary testing in the context of complex data types must be regarded as workable.

Despite the achievements the project clearly identified the need for research in the areas of flexible test selection policies and their application in the context of *hep*TEST.

Chapter 7

Conclusions and Further Work

The aim of the work presented in this thesis is to improve the current testing process through automation. We wanted to introduce systematic test creation to industrial testing. In this chapter we examine the extent to which this aim has been achieved and suggest the form and content of future related work. In Section 7.1 we summarise the contribution of the thesis and analyse the limitations of the proposal. The remainder of this chapter introduces thoughts on how to redeem these.

7.1 Contribution of the Thesis

The hypothesis underpinning this thesis is that automated generation of software-based system tests can be used to improve the testing process and cost-effectiveness of test creation by "lifting" traditional testing methodology to an abstract level. At the abstract level we can use a formal system to facilitate the automatic computation of test inputs and expected outcomes. As a result we can use the traditional testing ideas within the higher level of a formal system for test case generation.

In this thesis the following assertions are made:

- Specifications of system requirements can be exploited for automated test case generation. Not all specifications lend themselves to a high degree of test automation. A constructive approach to the definition of system properties is needed. We proposed to make use of a specialised specification language which we called - *hep*SPEC. The language *hep*SPEC is a modular parameterisable algebraic specification language built on the foundations proposed by Reichel (1987) and modified with automated

system test generation in mind. Thus it contains certain restrictions, like primitive recursion which were discussed in detail in Chapter 4.4.

- The need to generate tests automatically leads us to a formal system for mechanical transformation and computation. We make use of the formal system introduced with *hep*SPEC. The mechanisation is based on term rewriting and is achieved through an adaptation of the LoFT approach reported by Marre (1995).

- The aim was to use traditional strategies at the abstract level of the formal system. In most traditional strategies test case selection is based upon an order, e.g. minimum and maximum. We had to find an order which could be created mechanically and so we used in this thesis the partial order over terms based on the sub-term relation as the ordering criterion for test strategies on test data.

- Test case generation needs to overcome the "abstraction gap" introduced by the use of an abstract specification. The tests computed in the abstract formal system cannot be applied in their abstract form to the implementation. The higher the level of abstraction in the specification the wider the gap between abstract test and implementation. We identified this problem and proposed the use of syntactic and semantic homomorphisms. These homomorphisms are introduced through grammars and are integrated into the formal framework. This transformation step has been called test case reification and is automated as well. The tester defines the grammars and the test reification is integrated into the automated test generation process.

- Test case execution can be supported through these homomorphisms as well. By changing only the homomorphisms we can support a move from manual to automated test execution during the development cycle. Because we leave the specification unchanged the same abstract tests are used to create the concrete tests for automated execution providing a true regression test.

- Test result validation can be eased if an expected outcome is provided. We proposed to use the system requirements specification to compute this outcome and to transform it into the desired format for test validation. This frees the tester from the tasks to provide an expected test outcome manually or to use a different approach to compute it. With the transformation the possibility is given to adopt the most applicable tool or method for test result validation. This will remain to be a testers responsibility.

The contribution of the thesis is validated through two examples. One is a simple example used to illustrate the technology. The other is an industrial case study adopted from the aviation industry to analyse the practical suitability of the approach.

It is claimed that the work reported here makes the following contributions in the area of functional system testing:

- The work takes a practical approach to automated system test generation. Traditional testing methods are "lifted" to an abstract level.

- The work provides a system description language – *hep*SPEC– which lends itself to automated test generation. This specification language is based on work by Reichel (1987) and Kaphengst and Reichel (1971) who proposed a novel approach to domain specification. This method is very useful for describing finite domains like those we encountered in our examples. We have adopted and modified this work with special considerations towards automated system test generation.

- Traditional methods of test selection are supported through a partial order over terms. This order is used to select test cases.

- Test reification is the mapping from abstract to concrete test cases which is described with grammars. Because grammars are very closely linked to our specification concept we do not need to switch the formalism during test case generation.

- A variety of execution models like manual, semi-automatic, or automatic can be supported with relatively little effort through test reification. This has the benefit that a switch from a manual to an automated or a switch between different tools can be achieved with more ease.

- Test result validation is supported by our approach through computation of an expected outcome. The automatic transformation of the abstract outcome into a desired format for use with existing test result validation utilities is achieved again by grammars.

The fact that all transformations and computations are achieved within the same formal system has to be an advantage. Not only is analysis easier but also other methods like proof can be incorporated into this framework.

The proposed system is limited by two factors:

- *The system properties which are testable in this framework.* Only systems which can be described within the limits of our specification language *hep*SPEC can be tested. The restrictions of *hep*SPEC and the difficulty to describe properties relevant in real-time and reactive systems limit its applicability. This leads to the undesirable case where certain important properties can not be included in the system test. Although it might not be possible to describe all systems within a single framework we might wish to extend the number of possible systems.

- *The test strategies implemented by this framework.* We implemented only a small number of strategies for this thesis. In practice more strategies might be used and the tester may wish to implement his or her own. With the current system a single strategy is used throughout the entire system. It is not a realistic approach when we target inhomogeneous systems. When parts of the system are developed in hardware and in software then this differences should be reflected in the test strategies for the different parts.

To address this weaknesses and deficiencies in this work we propose enhancements and further research.

7.2 Enhancements and Future Research Directions

Enhancements to the framework developed in this thesis are considered in two groups. First enhancements to the theoretical foundations in order to make the approach applicable to wider classes of computing systems. Secondly the technical enhancements which are aimed at improving the practicality of the approach. Concluding we provide ideas for future research.

7.2.1 Theoretical Enhancements

On the theoretical side we restricted the operations to primitive recursion. Our goal was to ensure computability. Only then we could always provide an expected outcome. This restriction may prove to be excessive. Kaphengst (1981) pointed to the interesting class of functions like primitive recursive, which we made use of, and others. These others may be an interesting alternative and a way to more expressiveness in *hep*SPEC. New research needs to be conducted to find ways to weaken it and to retain computability.

There are certain concepts which cannot be specified in *hep*SPEC. For example real-time constraints are not specifiable in *hep*SPEC. There are algebraic methods which aim to include such concepts(Baum et al., 1999; Deutsch and Kaplan, 1992). Enhancements could investigate if our approach could be extended by those concepts. In addition to extending the formal specification base one has to investigate what tests could be derived from such a specification. Two kinds of features make real-time systems especially difficult to test. Some real-time systems are characterised through response times of operations. An operation has to return the answer within 2 ms, for example. Other real-time system operations return different values at different times. One can say that the function has an additional parameter time which is consulted in the computation of the outcome. These characteristics are an important part of real-time systems and need to be tested. Kopetz (1997) refers to the difficulties involved when he talks about the problem of interference when making observations in real-time systems. He points out that making the observation itself changes the behaviour of the system under observation. The question to be answered is: How to derive relevant tests — including the expected outcome — for real-time systems?

A third improvement would be the creation of a test strategy language. A tester would write a test strategy specification for the system test using this language. It could allow the tester to provide custom test strategies for different parts of the system. This should cater for the fact that a system is not built in a homogeneous way but consists of many components which are to be tested differently. An initial approach is illustrated in Section 5.9 were we discussed other testing strategies. There we proposed to use the structure of the specification to generate sub-domains. A test strategy language could name functions and specify the depth of decomposition. A similar method is proposed by Marre (1995) within the concept of LoFT. In discussions with Marre it became apparent that such a technology is not very transparent. It is very difficult for the tester to estimate the impact of the directives. The same would be true in the context of *hep*SPEC. Furthermore this approach takes a very localised view of operations. A fundamental operation is likely to be used in very different contexts. As indicated above, these contexts might be located in different parts of the system and could require different treatment. Recent developments in a very different area might lend us ideas. The primary purpose of XPath (Consortium, 1999) is to address parts of an XML document. In support of this primary purpose, it also provides basic facilities for manipulation of strings, numbers and booleans. XPath operates on the abstract, logical structure of an XML document, rather than its surface syntax. In addition to its use for addressing, XPath is also designed so that it has a natural subset

that can be used for matching. With the use of XPath constructs we could express the problem described earlier and it makes XPath a worthwhile candidate for investigations.

7.2.2 Technical Enhancements

Our implementation is based on term-rewriting systems. But with the recent developments in model-checkers (Somenzi, 1999) and their implementations, e.g. SPIN (Holzmann, 1997), these could be a basis for implementation. Model-checking relies on the efficiency of the binary decision diagram (BDD) structure to represent a very large domain space. In contrast term-rewriting systems hold only a very small part of the solution space in memory at a time. The possible advantage of model-checking could be that the domain space is spanned and could be examined more closely. This would mean that model-checking technology might provide a means for searching the space of possible tests and implementing more elaborate test strategies. These test strategies could take advantage of the context and might use the model as a means of test coverage. It should be investigated whether these possibilities have any impact on practical test case generation. An implementation and a comparative study should give the answer to the suitability of model checkers as a basis for implementation of a test generation engine.

A tool set which allows the creation and animation of very large *hep*SPEC-specifications needs to be created. We have already started with the basic components, like basic editor and syntax checker (Hayes, 1997). For the test creation process we used ECLiPSe (Aggounand et al., 1999), LoFT (Bernot et al., 1991b) and made our own adoptions. But there is room for major improvements. An initial step into this direction is taken by Walker (2001), who has created a visual development environment (VDE) for *hep*SPEC. This VDE, called Visual *hep*SPEC, would allow the visual creation of specifications. Further projects may follow. This VDE could be enhanced by an intelligent editor using the ideas by Antoy (1989) for design strategies of algebraic specifications. But also the connection to term-rewriting needs improvements. A term-rewriting engine that supports specifically *hep*SPEC and *hep*TEST needs to be developed. The requirements would include fast animation of *hep*SPEC specifications and test case generation. The execution speed is vital if very large specifications are considered.

In specifying the industrial example we found that certain concepts could be formalised and reused later. This induced the idea of a concept library which is developed for a particular problem domain in industry. Elements of such a library would be sub-specifications, meaningful and useful specifications at a granularity which eases reuse. For the defini-

tion of those concepts we could borrow some of the ideas in Reichel (1987, Chapter 4.2). Such a library should ease the creation of *hep*SPEC-specifications. The manipulation and construction of libraries should be possible within Visual *hep*SPEC.

In addition such a library could contain different interchangeable specification modules for problem groups. Similar to the concept of object-orientation these modules would have a common interface but could represent different internal aspects. This might improve the execution times during modelling or computation times during expected outcome generation. It would be the tester's responsibility to choose an adequate description.

A library for test reification would be desirable as well. Such a library would consist of patterns for manual test execution or test script languages for different commercial products. This could increase the independence from a particular test execution tool.

Another technical enhancement would be a possibility to visualise the test space. Such a visualisation component has to deal with very large amounts of data. It also needs to present it in a manner such that an engineer could use it to access the suitability of a testing strategy. No steps into this direction have been made yet but ideas could be borrowed from other sources, e.g. Sanders (1996).

All of these technical enhancements should be combined in an integrated development environment.

7.2.3 Impact on other Research Areas

This work only provides the frame for future research. Our goal was to provide an enhancement to existing test technologies. Therefore we aimed to build a basis for automated test generation which follows a pre-selected test strategy. We argued that the test strategy is best selected by an experienced tester who can use his knowledge to determine which strategy is most appropriate. Future research could investigate this closely. Using our automated test generation approach and different test strategies the effectiveness of a particular strategy with respect to a class of problems could be measured. This new research could give rise to new test strategies.

We tried to take a holistic approach and to support most steps of the testing process in our approach. There are other factors outside of testing which need to be considered as well. For example in industry requirements traceability has become an important method to ensure quality of systems (Finkelstein and Stevens, 1997). Requirements are traced through the entire development cycle and also through testing. Each requirement is associated with a test set which covers it. It would be desirable that automated test generation is included

into requirements traceability. But this is not straightforward and would require further investigations.

Fundamentally testing cannot be seen in isolation. Other methods for quality assurance exist and future research could seek to answer the question of how these methods could co-operate more efficiently. In the case of automated test generation in conjunction with proof the question could be how is a proof affected if test data could be generated to cover a certain property. In the other it could ask, which tests can be omitted if a certain property is proven. This opens a whole new chapter in the area of quality assurance. Our contribution, although small, could be used to verify or assess such propositions. Using an example with injected faults test could be created automatically based on our approach using a strategy incorporating proven properties. Measuring the effectiveness of the strategy in finding faults or indentifying those types of faults which were not found could help to bring this area of research a step forward.

In this thesis we aimed to show how traditional, industrially accepted testing methodologies could be "lifted" to a formal specification level in order to support automated test generation. We hope that the ideas will help to put automated support for testing on a more formal basis and will allow it to be used alongside other verification and validation methods in the future.

Appendix A

Triangle example - specification

```
BOOL is generation
        sorts Bool
        oprn true → Bool
             false → Bool

     with definition
        oprn and(Bool,Bool) → Bool
              or(Bool,Bool) → Bool
             not(Bool) → Bool

        axioms b:Bool
         and(false,b) = false
          and(true,b) = b
           or(true,b) = true
          or(false,b) = b
            not(true) = false
           not(false) = true
end BOOL

SIDE is BOOL with generation
        sorts Nat
        oprn zero → Nat
             succ(Nat) → Nat

     with definition
        oprn add(Nat,Nat) → Nat
             eq(Nat,Nat) → Bool
             le(Nat,Nat) → Bool

        axioms a,b:Nat
```

```
                add(zero,b) = b
              add(succ(a),b) = succ(add(a,b))
               eq(zero,zero) = true
             eq(succ(a),zero) = false
             eq(zero,succ(b)) = false
          eq(succ(a),succ(b)) = eq(a,b)
               le(zero,zero) = false
             le(succ(a),zero) = false
             le(zero,succ(b)) = true
          le(succ(a),succ(b)) = le(a,b)
end SIDE

TRIANGLE is SIDE with definition
        oprn validTriangle(a,b,c) → Bool
        axioms a,b,c:Nat
               validTriangle = and(
                       and(
                       le(a,add(b,c)),
                       le(b,add(c,a))),
                       le(c,add(a,b)))

        with generation
        sorts Triangle
        oprn scalene → Triangle
             isosceles → Triangle
             equilateral → Triangle

        with definition
        oprn triangle(a:Nat,b:Nat,c:Nat
               iff validTriangle(a,b,c) = true
               ) → Triangle

        axioms a:Nat,b:Nat,c:Nat
        if validTriangle(a,b,c) = true
            eq(a,b) = false
            eq(a,c) = false
            eq(b,c) = false
          then triangle(a,b,c) = scalene
        if validTriangle(a,b,c) = true
            or(or(and(eq(a,b),not(eq(a,c))),
                  and(eq(a,c),not(eq(a,b)))),
                  and(eq(b,c),not(eq(a,c)))) = true
          then triangle(a,b,c) = isosceles
```

```
        if validTriangle(a,b,c) = true
           eq(a,c) = true
           eq(a,b) = true
          then triangle(a,b,c) = equilateral
end TRIANGLE
```

Appendix B

Name store example - specification

```
Bool is generation
 sorts Bool
 oprn true → Bool
      false → Bool
end Bool

Nat is Bool with generation
  sorts Nat
  oprn 0 → Nat
       s(Nat) → Nat
 with definition
  oprn lt(Nat,Nat) → Bool

  axioms a,b:Nat
  lt(a,0) = false
  lt(0,s(b)) = true
  lt(s(a),s(b)) = lt(a,b)
end Nat

Member is Bool with generation
 sorts Member
 oprn
       name1 → Member      name7 → Member
       name2 → Member      name8 → Member
       name3 → Member      name9 → Member
       name4 → Member      name10 → Member
       name5 → Member      name11 → Member
       name6 → Member
```

```
 with definition
 oprn eqMember(Member,Member) → Bool

 axioms a,b: Member
      eqMember(a,a) = true
      if eqMember(a,b) = false
        then eqMember(b,a) = false

      eqMember(name1,name2) = false     eqMember(name2,name3) = false
      eqMember(name1,name3) = false     eqMember(name2,name4) = false
      eqMember(name1,name4) = false     eqMember(name2,name5) = false
      eqMember(name1,name5) = false     eqMember(name2,name6) = false
      eqMember(name1,name6) = false     eqMember(name2,name7) = false
      eqMember(name1,name7) = false     eqMember(name2,name8) = false
      eqMember(name1,name8) = false     eqMember(name2,name9) = false
      eqMember(name1,name9) = false     eqMember(name2,name10) = false
      eqMember(name1,name10) = false    eqMember(name2,name11) = false
      eqMember(name1,name11) = false    eqMember(name4,name5) = false
      eqMember(name3,name4) = false     eqMember(name4,name6) = false
      eqMember(name3,name5) = false     eqMember(name4,name7) = false
      eqMember(name3,name6) = false     eqMember(name4,name8) = false
      eqMember(name3,name7) = false     eqMember(name4,name9) = false
      eqMember(name3,name8) = false     eqMember(name4,name10) = false
      eqMember(name3,name9) = false     eqMember(name4,name11) = false
      eqMember(name3,name10) = false    eqMember(name6,name7) = false
      eqMember(name3,name11) = false    eqMember(name6,name8) = false
      eqMember(name5,name6) = false     eqMember(name6,name9) = false
      eqMember(name5,name7) = false     eqMember(name6,name10) = false
      eqMember(name5,name8) = false     eqMember(name6,name11) = false
      eqMember(name5,name9) = false     eqMember(name7,name8) = false
      eqMember(name5,name10) = false    eqMember(name7,name9) = false
      eqMember(name5,name11) = false    eqMember(name7,name10) = false
      eqMember(name8,name9) = false     eqMember(name7,name11) = false
      eqMember(name8,name10) = false    eqMember(name9,name10) = false
      eqMember(name8,name11) = false    eqMember(name9,name11) = false
      eqMember(name10,name11) = false
end Member


Store is Nat with Member with generation
 sorts Store
  oprn emptyStore → Store
```

```
with definition
 oprn inStore(Member,Store) → Bool
      numberOfMembers(Store) → Nat
      maxMembersInThisStore → Nat

with generation
 oprn enterInStore(x:Store,m:Member
       iff inStore(m,x)=false
           lt(numberOfMembers(x),maxMembersInThisStore) = true) → Store

with definition
 oprn removeFromStore(x:Store,m:Member
       iff inStore(m,x)=true) → Store

axioms x:Store, m,n:Member
     maxMembersInThisStore = s(s(s(s(s(s(s(s(s(s(0))))))))))
        // the maximum number of members in THIS store is 10

     inStore(m,emptyStore) = false
     if inStore(m,x) = false
        lt(numberOfMembers(x),maxMembersInThisStore) = true
      then inStore(m,enterInStore(x,m)) = true
     if inStore(m,x) = false
        lt(numberOfMembers(x),maxMembersInThisStore) = true
        eqMember(m,n) = false
      then inStore(n,enterInStore(x,m)) = inStore(n,x)

     numberOfMembers(emptyStore) = 0
     if inStore(m,x) = false
        lt(numberOfMembers(x),maxMembersInThisStore) = true
      then numberOfMembers(enterInStore(x,m)) = s(numberOfMembers(x))

     if inStore(m,x) = false
        lt(numberOfMembers(x),maxMembersInThisStore) = true
      then removeFromStore(enterInStore(x,m),m) = x
     if inStore(m,x) = true
        lt(numberOfMembers(x),maxMembersInThisStore) = true
        inStore(n,x) = false
        eqMember(m,n) = false
      then removeFromStore(enterInStore(x,n),m)
             = enterInStore(removeFromStore(x,m),n)
```

```
  with generation
  sorts File,FId
  oprn mkFile(FId, Store) → File
       output_data → FId
       other_data → FId

  with definition
  oprn fileExists(File) → Bool
       saveToFile(Store) → File
       loadFromFile(f:File
         iff fileExists(f)=true) → Store

  axioms x:Store,fid:FId
   fileExists(mkFile(output_data,x)) = true
   fileExists(mkFile(other_data,x)) = false

   saveToFile(x) = mkFile(output_data,x)

   if fileExists(mkFile(fid,x)) = true
    then loadFromFile(mkFile(fid,x)) = x
end Store

Grammar
   syn : Sentence;

   Sentence : CommandSeq [s] [nl] [q] nl]
     means Sentence = CommandSeq;


   CommandSeq : [s] [nl]
     means ComandSeq = emptyStore;

   CommandSeq : CommandSeq [i] [nl] Name [nl]
     where inStore(Name,CommandSeq1) = false
           lt(numberOfMembers(CommandSeq1),maxMembersInThisStore) = true
     means CommandSeq0 = enterInStore(CommandSeq1 ,Name);

   CommandSeq : CommandSeq [d] [nl] Name [nl]
     where inStore(Name,CommandSeq1) = true
     means CommandSeq0 = removeFromStore(CommandSeq1 , Name);
```

```
Name : [9]
  means Name = name1;
     | [$DF%x]
  means Name = name2;
     | [Smith]
  means Name = name3;
     | [ Smith]
  means Name = name3;
     | [Smith ]
  means Name = name3;
     | [smith]
  means Name = name3;
     | [SMITH]
  means Name = name3;
     | [aRwIPeD]
  means Name = name4;
     | [S58OmLM]
  means Name = name5;
     | [cwQ]
  means Name = name6;
     | [4swu]
  means Name = name7;
     | [CD41Y]
  means Name = name8;
     | [Uhdl]
  means Name = name9;
     | [VeryLongNameWithCharacters0123456789012345678]
  means Name = name10;
     | [&yBö]
  means Name = name11;



Result :
  means Result = emptyStore;

Result : Result Name [nl]
  where inStore(Name,Result1) = false
        lt(numberOfMembers(Result1),maxMembersInThisStore) = true
  means Result0 = enterInStore(Result1,Name);
```

```
   Bool: [YES, name is in store]
     means Bool = true;
   Bool: [NO, name is not in store]
     means Bool = false;
end Grammar
```

Appendix C

BCS case study - specification

```
Bool is generation
 sorts Bool
 oprn True → Bool
      False → Bool
end Bool

Nat is Bool with generation
 sorts Nat
 oprn
  0 → Nat
  succ(Nat) → Nat
 with definition
 oprn lt(Nat,Nat) → Bool
      leq(Nat,Nat) → Bool
      add(Nat,Nat) → Nat
      sub(a:Nat,b:Nat iff leq(b,a) = True) → Nat
      prev(a:Nat iff lt(0,a) = True) → Nat
 axioms a,b:Nat
  lt(0,succ(b)) = True
  lt(a,0) = False
  lt(succ(a),succ(b)) = lt(a,b)
  leq(0,b) = True
  leq(succ(a),0) = False
  leq(succ(a),succ(b)) = leq(a,b)
  add(0,b) = b
  add(succ(a),b) = succ(add(a,b))
  sub(a,0) = a
  if leq(succ(b),succ(a)) = True
   then sub(succ(a),succ(b)) = sub(a,b)
  prev(succ(a)) = a
```

```
 with definition
  oprn
      10 → Nat
  mod_10(Nat)→ Nat
  div_10(Nat)→ Nat
 axioms m:Nat
  10 = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))))
  if lt(m,10) = True
   then mod_10(m) = m
  if lt(m,10) = False
   then mod_10(m) = mod_10(sub(m,10))
  if lt(m,10) = True
   then div_10(m) = 0
  if lt(m,10) = False
   then div_10(m) = succ(div_10(sub(m,10)))
end Nat

NatPair is Nat with generation
 sorts TNatPair

 oprn mkPair(Nat, Nat) → TNatPair

 with definition
  oprn carry(TNatPair) → Nat
       result(TNatPair) → Nat

 axioms m,n:Nat
   carry(mkPair(m,n)) = m
  result(mkPair(m,n)) = n

 with definition
  oprn add_3(Nat, Nat, Nat) → TNatPair

 axioms m,n,r,c:Nat
  if r = add(add(m,n),c)
   then add_3(m,n,c) = mkPair(div_10(r), mod_10(r))
end NatPair

DigitSeq is NatPair with generation
 sorts TDigitSeq

 oprn sq(n:Nat iff lt(n,10) = True) → TDigitSeq
      app(t:TDigitSeq,n:Nat iff lt(n,10) = True) → TDigitSeq
```

```
with definition
 oprn
  s_length(TDigitSeq) → Nat
  prefix(n:Nat,t:TDigitSeq iff lt(n,10) = True) → TDigitSeq
  lead_0(TDigitSeq) → Bool

axioms m,n:Nat,t:TDigitSeq
 if lt(m,10) = True
  then s_length(sq(m)) = succ(0)
 if lt(m,10) = True
  then s_length(app(t,m)) = succ(s_length(t))

 if lt(m,10) = True lt(n,10) = True
  then prefix(m,sq(n)) = app(sq(m),n)
 if lt(m,10) = True lt(n,10) = True
  then prefix(m,app(t,n)) = app(prefix(m,t),n)

 if lt(m,10) = True
  then lead_0(sq(m)) = False
 if lt(succ(m),10) = True
  then lead_0(prefix(succ(m),t)) = False
 lead_0(prefix(0,t)) = True

with definition
 oprn
 l_normalize(TDigitSeq, Nat) → TDigitSeq

axioms m,n:Nat, t:TDigitSeq
 if lt(n,10) = True
    lt(s_length(t),n) = False
  then l_normalize(t,n) = t
 if lt(n,10) = True
   lt(s_length(t), n) = True
   n = succ(m)
  then l_normalize(t,n) = prefix(0,l_normalize(t,m))

with definition
 oprn dplus(s1:TDigitSeq, s2:TDigitSeq, n:Nat
               iff s_length(s1) = s_length(s2)) → TDigitSeq
      s_lt(s1:TDigitSeq, s2:TDigitSeq
            iff s_length(s1) = s_length(s2)) → Bool
     s_leq(s1:TDigitSeq, s2:TDigitSeq
            iff s_length(s1) = s_length(s2)) → Bool
```

```
      s_mul(TDigitSeq, TDigitSeq) → TDigitSeq
      s_add(TDigitSeq, TDigitSeq) → TDigitSeq
      sprev(s1:TDigitSeq
            iff s_lt(l_normalize(sq(0),s_length(s1)),s1) = True)
              → TDigitSeq
      trunc_3(TDigitSeq) → TDigitSeq

axioms m,n,c:Nat, r:TNatPair, s1,s2,s3:TDigitSeq
  if lt(m,10) = True
     lt(n,10) = True
     r = add_3(m,n,c)
     carry(r) = 0
   then dplus(sq(m),sq(n),c) = sq(result(r))

  if lt(m,10) = True
     lt(n,10) = True
     r = add_3(m,n,c)
     leq(carry(r),0) = False
   then dplus(sq(m),sq(n),c) = app(sq(carry(r)),result(r))

  if lt(m,10) = True
     lt(n,10) = True
     r = add_3(m,n,c)
     s_length(s1) = s_length(s2)
   then dplus(app(s1,m),app(s2,n),c) = app(dplus(s1,s2,carry(r)),result(r))

  if lt(m,10)=True lt(n,10)=True
   then s_leq(sq(m),sq(n)) = leq(m,n)
  if leq(m,n) = False s_length(s1) = s_length(s2)
      lt(m,10) = True lt(n,10) = True
   then s_leq(prefix(m,s1),prefix(n,s2)) = False
  if lt(m,n) = True s_length(s1) = s_length(s2)
      lt(m,10) = True lt(n,10) = True
   then s_leq(prefix(m,s1),prefix(n,s2)) = True
  if s_length(s1) = s_length(s2) lt(m,10) = True
   then s_leq(prefix(m,s1),prefix(m,s2)) = s_leq(s1,s2)

  if lt(m,10)=True lt(n,10)=True
   then s_lt(sq(m),sq(n)) = lt(m,n)
  if leq(m,n) = False s_length(s1) = s_length(s2)
      lt(m,10) = True lt(n,10) = True
   then s_lt(prefix(m,s1),prefix(n,s2)) = False
```

```
   if lt(m,n) = True s_length(s1) = s_length(s2)
       lt(m,10) = True leq(n,10) = True
    then s_lt(prefix(m,s1),prefix(n,s2)) = True
   if s_length(s1) = s_length(s2) lt(m,10) = True
    then s_lt(prefix(m,s1),prefix(m,s2)) = s_lt(s1,s2)

   if succ(m)=n
    then sprev(sq(n)) = sq(m)
   sprev(app(sq(succ(0)),0))=
     sq(succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))))
   if succ(m)=n
    then sprev(app(s1,n)) = app(s1,m)
   sprev(app(s1,0)) =
     app(sprev(s1),succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))))

   s_add(s1,s2) =
     dplus(l_normalize(s1,s_length(s2)),l_normalize(s2,s_length(s1)),0)

   s_mul(s1,sq(0)) = sq(0)
   if sq(succ(n)) = s2 lt(succ(n),10) = True
    then s_mul(s1,s2) = s_add(s1,s_mul(s1,sq(n)))
   if app(s3,n) = s2 lt(n,10) = True
    then s_mul(s1,s2) = s_add(s1,s_mul(s1,sprev(s2)))

   if leq(s_length(s1),succ(succ(succ(0)))) = True
     then trunc_3(s1) = sq(0)
   if leq(s_length(app(app(app(s1,m),n),c)),succ(succ(succ(0)))) = False
     then trunc_3(app(app(app(s1,m),n),c)) = s1
end DigitSeq

IntegerArithmetic is DigitSeq with generation
 sorts TIntNumber

 oprn  int(s1:TDigitSeq iff lead_0(s1) = False) → TIntNumber

 with definition
 oprn
  i_add(TIntNumber, TIntNumber) → TIntNumber
  i_mul(TIntNumber, TIntNumber) → TIntNumber
  i_leq(TIntNumber, TIntNumber) → Bool
   i_lt(TIntNumber, TIntNumber) → Bool
```

```
 axioms s1,s2:TDigitSeq,i1:TIntNumber
    if lead_0(s1) = False
       lead_0(s2) = False
     then i_add(int(s1),int(s2)) =
        int(dplus(l_normalize(s1,s_length(s2)),
                              l_normalize(s2,s_length(s1)),0))

    if lead_0(s1) = False lead_0(s2) = False
     then i_mul(int(s1),int(s2)) = int(s_mul(s1,s2))

    if lead_0(s1) = False
       lead_0(s2) = False
     then i_leq(int(s1),int(s2)) =
          s_leq(l_normalize(s1,s_length(s2)),l_normalize(s2,s_length(s1)))

    if lead_0(s1) = False
       lead_0(s2) = False
     then i_lt(int(s1),int(s2)) =
          s_lt(l_normalize(s1,s_length(s2)),l_normalize(s2,s_length(s1)))
end IntegerArithmetic

DecimalArithmetic is DigitSeq with generation
 sorts TDecNumber
 oprn ddot(s1:TDigitSeq iff  lead_0(s1) = False) → TDecNumber

 with definition
  oprn d_add(TDecNumber, TDecNumber) → TDecNumber
       d_mul(TDecNumber, TDecNumber) → TDecNumber
       d_leq(TDecNumber, TDecNumber) → Bool
       d_lt(TDecNumber, TDecNumber) → Bool

 axioms s1,s2:TDigitSeq,d1:TDecNumber
    if lead_0(s1) = False
       lead_0(s2) = False
     then d_add(ddot(s1),ddot(s2)) =
        ddot(dplus(l_normalize(s1,s_length(s2)),
                              l_normalize(s2,s_length(s1)),0))
    if lead_0(s1) = False lead_0(s2) = False
     then d_mul(ddot(s1),ddot(s2)) = ddot(trunc_3(s_mul(s1,s2)))
    if lead_0(s1) = False
       lead_0(s2) = False
     then d_leq(ddot(s1),ddot(s2)) =
          s_leq(l_normalize(s1,s_length(s2)),l_normalize(s2,s_length(s1)))
```

```
    if lead_0(s1) = False
       lead_0(s2) = False
     then d_lt(ddot(s1),ddot(s2)) =
          s_lt(l_normalize(s1,s_length(s2)),l_normalize(s2,s_length(s1)))
end DecimalArithmetic

Char is Bool with generation
   sorts Char
   oprn ca → Char
        cb → Char
  with definition
   oprn eqChar(Char,Char) → Bool
   axioms c: Char
   eqChar(c,c) = True
   eqChar(ca,cb) = False eqChar(cb,ca) = False
end Char

String is Char with Nat with generation
   sorts String

   oprn  newString → String
         appChar(String,Char) → String

  with definition
   oprn strlength(String) → Nat
        prefixChar(Char,String) → String
        eqString(String,String) → Bool

   axioms s,s1:String, c,c1:Char
     strlength(newString) = 0
     strlength(appChar(s,c)) = succ(strlength(s))
     prefixChar(c,newString) = appChar(newString,c)
     prefixChar(c,appChar(s,c1)) = appChar(prefixChar(c,s),c1)
     eqString(newString,newString) = True
     eqString(appChar(s,c),appChar(s1,c)) = eqString(s,s1)
     if eqChar(c,c1) = False
      then eqString(appChar(s,c),appChar(s1,c1)) = False
end String

WaypointID is String with definition
   oprn n3 → Nat
   axioms n3 = succ(succ(succ(0)))
```

```
  with generation
   sorts WaypointID
   oprn newWaypointID(s:String
                  iff leq(strlength(s),n3) = True) → WaypointID
end WaypointID

Longitude is DecimalArithmetic with definition
   oprn n180_0 → TDecNumber

   axioms
     n180_0 = ddot(app(app(app(app(app(sq(succ(0)),
          succ(succ(succ(succ(succ(succ(succ(succ(0))))))))),0),0),0),0))


  with generation
   sorts Longitude, Direction

   oprn newLongitude(d:Direction,n:TDecNumber
                  iff d_leq(n,n180_0)= True) → Longitude
        east → Direction
        west → Direction
end Longitude

Latitude is DecimalArithmetic with definition
   oprn n90_0 → TDecNumber
        n0_0 → TDecNumber
   axioms
     n90_0 = ddot(app(app(app(app(
              sq(succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))))
                   ,0),0),0),0))
     n0_0 = ddot(app(app(app(sq(0),0),0),0))

  with generation
   sorts Latitude, Hemisphaere

   oprn north → Hemisphaere
        south → Hemisphaere
        newLatitude(h:Hemisphaere,n:TDecNumber
                  iff d_leq(n,n90_0) = True) → Latitude
end Latitude

Elevation is IntegerArithmetic with definition
   oprn n10000 → TIntNumber
```

```
    axioms n10000 = int(app(app(app(app(app(
                                sq(succ(0)),0),0),0),0),0),0))
  with generation
    sorts Elevation
    oprn newElevation(n:TIntNumber iff i_leq(n,n10000)= True ) → Elevation
end Elevation

Waypoint is WaypointID with Longitude with Latitude
    with Elevation with generation

    sorts Waypoint

    oprn mkWaypoint(WaypointID,Longitude,Latitude,Elevation) → Waypoint

  with definition
    oprn getWaypointID(Waypoint) → WaypointID
         getWaypointLongitude(Waypoint) → Longitude
         getWaypointLatitude(Waypoint) → Latitude
         getWaypointElevation(Waypoint) → Elevation

    axioms waypointID: WaypointID, longitude: Longitude,
           latitude: Latitude, elevation: Elevation
     getWaypointID(mkWaypoint(waypointID,longitude,latitude,elevation))
                  = waypointID
     getWaypointLongitude(mkWaypoint(waypointID,longitude,latitude,elevation))
                  = longitude
     getWaypointLatitude(mkWaypoint(waypointID,longitude,latitude,elevation))
                  = latitude
     getWaypointElevation(mkWaypoint(waypointID,longitude,latitude,elevation))
                  = elevation
end Waypoint

Description is String with definition
    oprn n3 → Nat
    axioms n3 = succ(succ(succ(0)))
  with generation
    sorts Description
    oprn mkDescription(s:String iff leq(strlength(s),n3) = True)
            → Description
end Description
```

```
Modulation is generation
   sorts Modulation
   oprn AM → Modulation
        FM → Modulation
end Modulation

Frequency is DecimalArithmetic with definition
   oprn n1000_0 → TDecNumber
            n6 → Nat
   axioms
     n1000_0 = ddot(app(app(app(app(app(app(sq(succ(0)),0),0),0),0),0),0),0))
          n6 = succ(succ(succ(succ(succ(succ(0))))))

  with generation
   sorts Frequency

   oprn newFrequency(n:TDecNumber iff d_leq(n,n1000_0)= True) → Frequency
end Frequency

RadioCode is Description with Modulation with Frequency with generation
   sorts RadioCode

   oprn mkRadioCode(Description,Modulation,Frequency) → RadioCode

  with definition
   oprn getRadioCodeDescription(RadioCode) → Description
        getRadioCodeModulation(RadioCode) → Modulation
        getRadioCodeFrequency(RadioCode) → Frequency

   axioms desc: Description, mod: Modulation, freq: Frequency
     getRadioCodeDescription(mkRadioCode(desc,mod,freq)) = desc
     getRadioCodeModulation(mkRadioCode(desc,mod,freq)) = mod
     getRadioCodeFrequency(mkRadioCode(desc,mod,freq)) = freq
end RadioCode

WaypointSeq is Waypoint with generation
   sorts WaypointSeq
  with definition
   oprn lengthOfWaypointSeq(WaypointSeq) → Nat
        n12 → Nat
   axioms
   n12 =
    succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))))))
```

```
  with generation
   oprn emptyWaypointSeq → WaypointSeq
        appendWaypoint(wps:WaypointSeq,wp:Waypoint
          iff lt(lengthOfWaypointSeq(wps),n12) = True) → WaypointSeq

  with definition
   axioms wps,wps1:WaypointSeq, wp:Waypoint
    lengthOfWaypointSeq(emptyWaypointSeq) = 0
    if wps1 = appendWaypoint(wps,wp)
      then lengthOfWaypointSeq(appendWaypoint(wps,wp))
           = succ(lengthOfWaypointSeq(wps))
end WaypointSeq

RadioCodeSeq is RadioCode with generation
   sorts RadioCodeSeq
  with definition
   oprn lengthOfRadioCodeSeq(RadioCodeSeq) → Nat
        n2 → Nat
   axioms n2 = succ(succ(0))
  with generation
   oprn emptyRadioCodeSeq → RadioCodeSeq
        addRadioCode(rs:RadioCodeSeq, r: RadioCode
          iff lt(lengthOfRadioCodeSeq(rs),n2) = True) → RadioCodeSeq
  with definition
   axioms rs,rs1: RadioCodeSeq, r: RadioCode
     lengthOfRadioCodeSeq(emptyRadioCodeSeq) = 0
     if rs1 = addRadioCode(rs,r)
       then lengthOfRadioCodeSeq(addRadioCode(rs,r))
            = succ(lengthOfRadioCodeSeq(rs))
end RadioCodeSeq

Vehicle is String with WaypointSeq
         with RadioCodeSeq with generation

   sorts Vehicle, VehicleId
   oprn newVehicle(VehicleId,WaypointSeq,RadioCodeSeq) → Vehicle
        newVehicleId(s:String
                     iff lt(0,strlength(s)) = True
                         leq(strlength(s),n3) = True) → VehicleId

  with definition
   oprn getVehicleId(Vehicle) → VehicleId
        eqVehicleId(VehicleId,VehicleId) → Bool
```

```
   axioms aId:VehicleId,wps:WaypointSeq,rcs:RadioCodeSeq, s1,s2:String
     getVehicleId(newVehicle(aId,wps,rcs)) = aId
     eqVehicleId(newVehicleId(s1),newVehicleId(s2)) = eqString(s1,s2)
end Vehicle

Mission is Vehicle
  with generation
   sorts Mission, MissionId
   oprn newMission(MissionId) → Mission
        mkMissionId(s:String
                    iff lt(0,strlength(s)) = True
                        leq(strlength(s),10)= True) → MissionId
  with definition
   oprn numOfVehicles(Mission) → Nat
  with generation
   oprn addVehicleMission(m:Mission, a:Vehicle
               iff leq(numOfVehicles(m),10) = True) → Mission

  with definition
   oprn getMissionId(Mission) → MissionId
        eqMissionId(MissionId,MissionId) → Bool
        existVehicle(Mission,VehicleId) → Bool
   axioms m,m1:Mission, mId,mId1,mId2:MissionId,s1,s2:String,
          ac:Vehicle, aId:VehicleId
     getMissionId(newMission(mId)) = mId
     eqMissionId(mId,mId) = True
     if leq(strlength(s1),10) = True
        leq(strlength(s2),10) = True
      then eqMissionId(mkMissionId(s1),mkMissionId(s2)) = eqString(s1,s2)
     numOfVehicles(newMission(mId)) = 0
     if m1 = addVehicleMission(m,ac)
      then numOfVehicles(addVehicleMission(m,ac)) = succ(numOfVehicles(m))
     existVehicle(newMission(mId),aId) = False
     if getVehicleId(ac) = aId
      then existVehicle(addVehicleMission(m,ac),aId) = True
     if eqVehicleId(getVehicleId(ac),aId) = False
      then existVehicle(addVehicleMission(m,ac),aId) = existVehicle(m,aId)
end Mission

MissionDB is Mission with generation
   sorts MissionDB
   oprn emptyMissionDB(Nat) → MissionDB
```

```
  with definition
   oprn numberOfMissions(MissionDB) → Nat
        maxNumOfMissions(MissionDB) → Nat
        existMission(MissionDB,MissionId) → Bool
  with generation
   sorts MissionIdSeq
   oprn insertMission(mdb:MissionDB,m:Mission
          iff lt(numberOfMissions(mdb),maxNumOfMissions(mdb))= True
              existMission(mdb,getMissionId(m)) = False
                    ) → MissionDB
        emptyMissionIdSeq → MissionIdSeq
        appendMissionId(MissionIdSeq,MissionId) → MissionIdSeq
  with definition
   oprn listMissionIds(MissionDB) → MissionIdSeq
        getMission(mdb:MissionDB,mId:MissionId
          iff existMission(mdb,mId)=True) → Mission
   axioms n:Nat, mdb:MissionDB, m,m1,m2:Mission, mId:MissionId
     existMission(emptyMissionDB(n),mId) = False
     if getMissionId(m) = mId
       then existMission(insertMission(mdb,m),mId) = True
     if eqMissionId(getMissionId(m),mId) = False
       then existMission(insertMission(mdb,m),mId) = existMission(mdb,mId)
     numberOfMissions(emptyMissionDB(n)) = 0
     numberOfMissions(insertMission(mdb,m)) = succ(numberOfMissions(mdb))
     maxNumOfMissions(emptyMissionDB(n)) = n
     maxNumOfMissions(insertMission(mdb,m)) = maxNumOfMissions(mdb)
     if getMissionId(m) = mId then getMission(insertMission(mdb,m),mId) = m
     if existMission(insertMission(mdb,m),mId) = True
        eqMissionId(getMissionId(m),mId)=False
      then getMission(insertMission(mdb,m),mId) = getMission(mdb,mId)
     listMissionIds(emptyMissionDB(n)) = emptyMissionIdSeq
     if lt(numberOfMissions(mdb),maxNumOfMissions(mdb))=True
        existMission(mdb,getMissionId(m)) = False
      then listMissionIds(insertMission(mdb,m))
           = appendMissionId(listMissionIds(mdb),getMissionId(m))
end MissionDB

Cartridge is String with WaypointSeq with RadioCodeSeq with generation
   sorts Cartridge
   oprn emptyCartridge → Cartridge
        mkCartridge(WaypointSeq,RadioCodeSeq) → Cartridge
end Cartridge
```

```
Units is generation
   sorts Units,Degree,Height
   oprn  newUnits(Degree,Height) → Units
         degree → Degree
         deg_min → Degree
         deg_min_sec → Degree
         ft → Height
         m → Height
  with definition
   oprn getDegree(Units) → Degree
        getHeight(Units) → Height
   axioms d:Degree, h:Height, u:Units
     getDegree(newUnits(d,h)) = d
     getHeight(newUnits(d,h)) = h
end Units

DataImage is Units with String with WaypointSeq
       with RadioCodeSeq with generation
   sorts DataImage
   oprn emptyDataImage → DataImage
        mkDataImage(WaypointSeq,RadioCodeSeq,Units) → DataImage

  with definition
   oprn changeUnitsWPS(WaypointSeq,Units) → WaypointSeq
        changeUnitsWP(Waypoint,Units) → Waypoint
        changeUnitDegree(TDecNumber,Units) → TDecNumber
        changeUnitHeight(TIntNumber,Units) → TIntNumber
   axioms wps:WaypointSeq,rcs:RadioCodeSeq,u:Units,wp:Waypoint,
       ident:WaypointID,dir:Direction,hem:Hemisphaere,long,lat,d:TDecNumber,
       el,h:TIntNumber,res:TDigitSeq

     changeUnitsWPS(emptyWaypointSeq,u) = emptyWaypointSeq
     changeUnitsWPS(appendWaypoint(wps,wp),u)
        = appendWaypoint(changeUnitsWPS(wps,u),changeUnitsWP(wp,u))

     changeUnitsWP(mkWaypoint(ident,newLongitude(dir,long),
           newLatitude(hem,lat),newElevation(el)),u)
       = mkWaypoint(ident,newLongitude(dir,changeUnitDegree(long,u)),
          newLatitude(hem,changeUnitDegree(lat,u)),
            newElevation(changeUnitHeight(el,u)))

     if getHeight(u) = ft
      then changeUnitHeight(h,u) = h
```

```
    if getHeight(u) = m
       int(res)=i_mul(h,int(app(app(app(sq(succ(succ(succ(0)))),
                succ(succ(0))),
                succ(succ(succ(succ(succ(succ(succ(succ(0))))))))),succ(0))))
     then changeUnitHeight(h,u) = int(trunc_3(res))

    if getDegree(u) = degree
     then changeUnitDegree(d,u) = d
    if getDegree(u) = deg_min
     then changeUnitDegree(d,u) = d
    if getDegree(u) = deg_min_sec
     then changeUnitDegree(d,u) = d
end DataImage

BCS is MissionDB with DataImage with Cartridge
     with Units with generation
    sorts BCS
    oprn mkBCS(MissionDB,DataImage) → BCS
   with definition
    oprn initBCS(MissionDB) → BCS
         existMissionInBCS(BCS,MissionId) → Bool
         existVehicleInBCS(bcs:BCS,mId:MissionId,aId:VehicleId
                           iff existMissionInBCS(bcs,mId) = True
                          ) → Bool
         notEmptyDataImage(BCS) → Bool
         chooseMission(bcs:BCS,mId:MissionId,aId:VehicleId
                       iff existMissionInBCS(bcs,mId)=True
                            existVehicleInBCS(bcs,mId,aId)=True
                      ) → BCS
         viewDataImage(bcs:BCS,u:Units
           iff notEmptyDataImage(bcs)=True) → DataImage
         loadCartridge(bcs:BCS
           iff notEmptyDataImage(bcs)=True) → Cartridge
         addWaypointToDataImage(BCS,Waypoint) → BCS
         addRadioCodeToDataImage(BCS,RadioCode) → BCS
    axioms mdb:MissionDB,mId:MissionId,aId:VehicleId,dataimage:DataImage,
           wps:WaypointSeq,wp:Waypoint,rcs:RadioCodeSeq,rc:RadioCode,
           u,new_u:Units

        initBCS(mdb) = mkBCS(mdb,emptyDataImage)
        existMissionInBCS(mkBCS(mdb,dataimage),mId) = existMission(mdb,mId)
        if existMission(mdb,mId)=True
         then existVehicleInBCS(mkBCS(mdb,dataimage),mId,aId)
```

```
            = existVehicle(getMission(mdb,mId),aId)
        if existMission(mdb,mId) = True
           existVehicle(getMission(mdb,mId),aId) = True
         then chooseMission(mkBCS(mdb,dataimage),mId,aId)
            = mkBCS(mdb,mkDataImage(wps,rcs,u))
        notEmptyDataImage(mkBCS(mdb,emptyDataImage)) = False
        notEmptyDataImage(mkBCS(mdb,mkDataImage(wps,rcs,u))) = True

        viewDataImage(mkBCS(mdb,mkDataImage(wps,rcs,u)),new_u)
           = mkDataImage(changeUnitsWPS(wps,new_u),rcs,new_u)
        loadCartridge(mkBCS(mdb,mkDataImage(wps,rcs,u)))
           = mkCartridge(wps,rcs)
        addWaypointToDataImage(mkBCS(mdb,mkDataImage(wps,rcs,u)),wp)
           = mkBCS(mdb,mkDataImage(appendWaypoint(wps,wp),rcs,u))
        addRadioCodeToDataImage(mkBCS(mdb,mkDataImage(wps,rcs,u)),rc)
           = mkBCS(mdb,mkDataImage(wps,addRadioCode(rcs,rc),u))
end BCS
```

Bibliography

Abderrahamane Aggounand et al. *ECLiPSe User Manual.* International Computers Limited and Imperial College London, release 4.2 edition, 1999.

A. V. Aho, Ravi Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1985.

Sergio Antoy. Design strategies for algebraic specifications. Technical Report TR-89-32, Virginia Polytechnic Inst. and State University, 1989.

Sergio Antoy and Dick Hamlet. Automatically checking an implementation against its formal specification. In *Irvine Software Symposium*, pages 29–48, 1992.

J. A. Bauer and A. B. Finger. Test plan generation using formal grammars. In *4th International Conference on Software Engineering*, pages 425–432, Long Beach, Ca., USA, 1979. IEEE Computer Society Press.

G. A. Baum, M. F. Frias, and T. S. E. Maibaum. A logic for real-time systems specification, its algebraic semantics, and equational calculus. *Lecture Notes in Computer Science*, 1548:91–105, 1999.

B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, 2nd edition, 1990.

Boris Beizer. *Black-Box Testing - Techniques for Functional Testing of Software and Systems.* John Wiley & Sons, Inc., 1995.

Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. A formal approach to software testing. In *Proceedings of the Second International Conference on Algebraic Methodology and Software Technology*, pages 163–170, Iowa City, Iowa, 1991a. The University of Iowa, Department of Computer Science.

Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, pages 387–405, 1991b.

Elaine J. Weyuker Bingchaing Jeng. A simplified domain-testing strategy. *ACM Transactions on Software Engineering and Methodology*, 3(3):254–270, 1994.

L. Bougé, N. Choquet, L. Fribourg, and M.-C. Gaudel. Test sets generation from algebraic specifications using logic programming. *Journal of Systems and Software*, 6(4):343–360, 1986.

L. Bouma and H. Walters. Implementing Algebraic Specifications. In J. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, chapter 5, pages 199–282. ACM Press, 1989.

C. J. Burgess. Software testing using an automatic generator of test data. In *Software Quality Management*, pages 541–556. Elsevier Science Publishers, 1993.

D. Carrington and P. Stocks. A tale of two paradigms: Formal methods and software testing. In J. P. Bowen and J. A. Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 51–68. Springer-Verlag, 1994.

K.-T. Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In ACM-SIGDA; IEEE, editor, *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 86–91, Dallas, TX, 1993. ACM Press.

World Wide Web Consortium. XML Path Language. W3C recommendation, World Wide Web Consortium, 1999.

G. Deutsch and S. Kaplan. Algebraic semantics of real-time process specifications. In Maurice Nivat, Charles Rattray, Teodor Rus, and Giuseppe Scollo, editors, *Proceedings of the Second International Conference on Algebraic Methodology and Software Technology*, Workshops in Computing, pages 83–97, London, 1992. Springer Verlag.

Jeremy Dick. Private communication, 2001.

Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, pages 268–284. Formal Methods Europe, Springer-Verlag, 1993. Lecture Notes in Computer Science 670.

M. Donat. Automating Formal Specification Based Testing. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97:Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, volume 1214 of *Lecture Notes in Computer Science*, pages 833–847, Lille, France, 1997. Springer-Verlag, Berlin.

Michael R. Donat. *A Discipline of Specification-Based Test Derivation.* PhD thesis, University of British Columbia, 1998.

M. Dyer. *The Cleanroom Approach to Quality Software Development.* John Wiley & Sons, New York, 1992.

Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, New York, N.Y., 1985.

E.W. Dijkstra. Notes on Structured Programming. In O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming.* Academic Press, 1972.

A. Finkelstein and R. Stevens. Requirements traceability. In *Proceedings: 3rd IEEE International Symposium on Requirements Engineering*, page 265. IEEE Computer Society Press, 1997.

S. Fujiwara, G. v. 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.

Marie-Claude Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, 1995.

Joseph Goguen and Razvan Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4:363–392, 1994.

J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, (10):27–52, 1978.

M. Hayes. Development of a Syntax Analyser for the OBGS Formal Specification Language. Master's thesis, University of Newcastle upon Tyne, 1997.

Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

Hans-Martin Hörcher and Jan Peleska. Using formal specifications to support software. *Software Quality Journal*, 4(4):309–327, 1995.

D. C. Ince. The automatic generation of test data. *The Computer Journal*, 30(1):63–69, 1987.

F. Ipate and M. Holcombe. A method for refining and testing generalised machine specifications. *International journal of computer mathematics.*, 68(3–4):197–219, 1998.

H. Kaphengst. What is computable for abstract data types? In Ferenc Gécseg, editor, *Proceedings of the 1981 International FCT-Conference on Fundamentals of Computation Theory*, volume 117 of *LNCS*, pages 173–181, Szeged, Hungary, 1981. Springer.

H. Kaphengst and Horst Reichel. Algebraische algorithentheorie. Technical Report WIB 1, VEB Robotron, Zentrum für Forschung und Technik, Dresden, 1971. In German.

Hermann Kopetz. *Real-Time Systems*, volume 395 of *The Kluwer International Series In Engineering And Computer Science*. Kluwer Academic Publishers, Boston, 1997.

Peter Liggesmeyer and Peter Rüppel. Die prüfung von objektorientierten Systemen. *OBJECTspektrum*, (6):68–78, 1996.

Timothy E. Lindquist and Joyce R. Jenkins. Test-case generation with IOGen. *IEEE Software*, 5(1):72–79, 1988.

B. Marre. LOFT: A tool for assisting selection of test data sets from algebraic specifications. *Lecture Notes in Computer Science*, 915:799–800, 1995.

Peter M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, 1990.

Christoph C. Michael, Gary E. McGraw, Michael A. Schatz, and Curtis C. Walton. Genetic algorithms for dynamic test data generation. Technical Report RSTR-003-97-11, RST Corporation, 1997.

Glenford J. Myers. *The Art of Software Testing*. Wiley - Interscience, New York, 1979.

N.P.Kropp, P.J.Koopman, and D.P.Siewiorek. Automated robustness testing of off-the-shelf software components. In *28th IEEE Symposium on Fault Tolerant Computing Systems (FTCS-28)*. ieee, 1998.

A. J. Offutt. Automatic test data generation. Technical Report SERC-TR-25-P, Software Engineering Research Centre, 1988.

Rózsa Péter. *Recursive Functions.* Academic Press, New York and London, 1967.

Robert M. Poston. Automated testing from object models. *Communications of the ACM*, 37(9):48–58, 1994.

Robert M. Poston. *Automating Specification-Based Software Testing.* IEEE Computer Society Press, 1996.

Paul Purdom. A sentence generator for testing parsers. *BIT*, 12:366–375, 1972.

H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras.* Oxford University Press, 1987.

Georg Sanders. *Visualisierungstechniken für den Compilerbau.* Pirrot Verlag, 66125 Saarbrücken, 1996.

F. Somenzi. Binary decision diagrams. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, volume 173 of *Series F: Computer and System Sciences*, part 3, pages 303–368. IOS Press, 1999.

Ian Sommerville. *Software Engineering.* Addison-Wesley, sixth edition, 2001.

L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Mass., 1986.

P. Stocks and D. Carrington. A Framework for Specification-Based Testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, 1996.

P. Stocks and D. A. Carrington. Deriving software test cases from formal specifications. In *6th Australian Software Engineering Conference*, pages 327–340, 1991.

P. Stocks and D. A. Carrington. Test template framework: A specification-based testing case study. In *Proc. International Symposium on Software Testing and Analysis (ISSTA'93)*, pages 11–18, 1993a.

P. Stocks and D. A. Carrington. Test templates: A specification-based testing framework. In *Proc. 15th International Conference on Software Engineering*, pages 405–414, 1993b.

Paul Strooper and Daniel Hoffman. Prolog testing of C modules. In Kazunori Saraswat, Vijay; Ueda, editor, *Proceedings of the 1991 International Symposium on Logic Programming (ISLP'91)*, pages 596–610, San Diego, CA, 1991. MIT Press.

A. von Mayrhauser, R. Mraz, and J. Walls. Domain based regression testing. In *Proceedings of the International Conference on Software Maintenance 1994*, pages 26–35, 1994a.

A. von Mayrhauser, J. Walls, and R. Mraz. Sleuth: A domain-based testing tool. In *International Test Conference*, pages 840–849, Altoona, Pa., USA, 1994b. IEEE Computer Society Press.

J. Walker. Visual *hep*SPEC. Final year project report, Dept of Computing Science, University of Newcastle upon Tyne, 2001.

Martin Wirsing. Algebraic specification. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 13, pages 675–788. The MIT Press, New York, NY, 1990.

N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14 (4):221–227, 1971.

Martin R. Woodward. Errors in algebraic specifications and an experimental mutation testing tool. *Software Engineering Journal*, 8(4):211–224, 1993.