Testing from Structured Algebraic Specifications:
The Oracle Problem

Patrícia D. L. Machado

Doctor of Philosophy
University of Edinburgh
2000

To Herman and to my Parents.

Abstract

Work in the area of specification-based testing has pointed out that testing can be effectively used to verify programs against formal specifications. The aim is to derive test information from formal specifications so that testing can be rigorously applied whenever full formal verification is not cost-effective. However, there are still several obstacles to be overcome in order to establish testing as a standard in formal frameworks. Accurate interpretation of test results is an extremely critical one.

This thesis is concerned with testing programs against structured algebraic specifications where axioms are expressed in first-order logic with equations, the usual connectives and quantifiers. The main issue investigated is the so-called *oracle problem*, that is, whether a decision procedure can be defined for interpreting the results of tests according to a formal specification. In this context, testing consists in checking whether specification axioms are satisfied by programs. Consequently, tests exercise operations referred to by the axioms and oracles evaluate the axioms according to the results produced by the tests.

The oracle problem for flat (unstructured) specifications often reduces to the problem of comparing two values of a non-observable sort, namely the equality problem, and also how to deal with quantifiers which may demand infinite test sets. Equality on non-observable sorts is interpreted up to behavioural equivalence with observational equivalence as an important special case. However, a procedure for implementing such a behavioural equality may be hard to define or even impossible. In this thesis, a solution to the oracle problem for flat specifications is presented which tackles the equality problem by using a pair of *approximate equalities*, one finer than behavioural equality and one coarser, and taking the syntactic position of quantifiers in formulae into account.

Additionally, when structured specifications are considered, the oracle problem can be harder. The reason is that specifications may be composed of parts over different signatures, and the structure must be taken into account in order to interpret test results according to specification axioms. Also, an implementation of hidden (non-exported) symbols may be required in order to check axioms which refer to them. Two solutions to the oracle problem for structured specifications are presented in this thesis based on a compositional and a non-compositional style of testing, namely

structured testing and *flat testing* respectively. Structured testing handles the oracle problem more effectively than flat testing and under fewer assumptions.

Furthermore, testing from structured specifications may require an approach which lies in between flat and structured testing. Therefore, based on normalisation of ordinary specifications, three normal forms are presented for defining a more practical and combined approach to testing and also coping more effectively with the oracle problem. The use of normal forms gives rise to a style of testing called *semi-structured testing* where some parts of the specification are replaced by normal forms and the result is checked using structured testing. Testing from normal forms can be very convenient whenever the original specification is too complex or oracles cannot be defined from it.

Acknowledgements

I wish to express my deep thanks to all those who have given me much encouragement to complete this thesis.

Many thanks to my supervisor, Don Sannella, for all his continuous support, invaluable advice and guidance. I have learned a lot from him on general research and specific subjects related to this thesis. I am very grateful to Stuart Anderson and Marie-Claude Gaudel for agreeing to be the examiners and for the comments they have made on the work presented in this thesis. Thanks also to Pascale Le Gall and her students for interesting discussions during my visit to Evry. I am also very grateful to all my colleagues and members of staff in the Division of Informatics, specially at LFCS, for the friendly and excellent environment for developing scientific work and the so many stimulating discussions. Thanks also to Gillian Hayes for being my mentor and the interesting conversations we had on general subjects. Many thanks to all friends I have made during my stay in Edinburgh. Special thanks to Edjard, Gisela, Alzira, Fausto, Tony and Mauricea.

The financial support for this thesis came from CNPq (Brazilian Research Council). Many thanks to the referees and members of staff who have always been so kind and attentive when responding to our needs. I have also been on leave of absence from DSC, Federal University of Paraíba, Brazil. Many thanks to all my colleagues there, specially Bernardo and Joseluce.

My eternal gratitude to my husband Herman for his everlasting love and support and also for so many interesting discussions and comments on the contents of this thesis. Also, to my devoted family, specially to my parents, Adelson and Perácia, who have waited so patiently for this thesis to appear. Many thanks to my parents-in-law, Cícero and Neuza, for all their support. And finally, many thanks to long-time friends in Brazil for their continuous affection and support.

Declaration

I declare that this thesis was composed by myself and that the work contained herein is my own, except where explicitly stated otherwise in the text. Part of the material in Chapters 3, 4 and 5 has already been published in [Machado, 1999], [Machado, 2000b] and [Machado, 2000a] respectively.

(*Patrícia D. L. Machado*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Thorough testing is the touchstone of reliability in quality assurance and control of modern production engineering."
C. A. R. Hoare

Contents

Testing is the process of exercising a program in order to uncover errors as part of verification and validation activities. Although it is usual to expend around 40 percent of total software project effort on testing, this is the least investigated and understood of all activities in the software development process [Goodenough and Gerhart, 1975, Hamlet, 1994, Pressman, 1994]. Besides, software development methods usually concentrate on analysis, design and implementation issues rather than verification and validation techniques. Even though testing can be applied in different stages of the software development process, it has usually been seen as an end activity which is performed only when the development process has finished and usually only the final code is tested. Nevertheless, it has been widely accepted that testing should be an intrinsic part of the development process and not only an add-on [Ould and Unwin, 1986, Douglas and Kemmerer, 1994, Bicarregui et al., 1997]. In order to establish testing as a reliable and effective verification technique, it is essential to develop well-founded methods and strategies [Hamlet, 1994, Bowen and Hinchey, 1995, Hoare, 1996].

Formal methods have been used to produce high integrity software in a number of application domains, including the industrial setting [Craigen et al., 1993, Hall, 1996, Gaudel and Woodcock, 1996, Saiedian, 1996]. However, full formal development, which involves formally specifying the system under consideration, refining the specification towards a more concrete representation and formally verifying each step in this refinement process, is rarely undertaken in practice. Also, even if this is the case, a substantial degree of testing is still needed, since both compilers and runtime environment may not be completely trustworthy [Dick and Faivre, 1993]. Moreover, formal proofs of correctness can give increasing confidence in the integrity of the system, but errors can still be found [Goodenough and Gerhart, 1975] which are mainly caused by errors and omissions in the specification and human/automated tool mistakes in the proofs as well. Furthermore, when large and complex systems are involved, full formal verification is either impossible or too costly while useful properties can be checked by testing [Bernot, 1991]. However, formal verification can be justified and strongly recommended when dependability is a necessary quality of safety-critical systems [Bowen and Stavridou, 1993], even though the real benefits remain controversial [Finney, 1996].

Testing is a classical and fundamental approach in practice to system verification. However, in order to detect correctness by testing, it is imperative to cover all possible ways of interacting with the system, usually an infinite and impractical activity. In fact, philosophers of science have pointed out that while testing can demonstrate the presence of "bugs", it cannot demonstrate their absence [Dijkstra, 1981]. Nevertheless, thorough testing can increase significantly confidence that software is correct, even if it cannot provide a total guarantee of correctness [Goodenough and Gerhart, 1975, Bernot, 1991, Mandrioli et al., 1995, Hoare, 1996]. Moreover, in specialised fields of engineering where mathematical techniques are employed for product verification, testing is required and applied as early as possible at all stations in the production line [Hoare, 1996].

The combination of formal methods and testing can help to produce high integrity systems in a cost-effective way with a partial or total guarantee of correctness [Carrington and Stocks, 1994, Bowen and Hinchey, 1995, Bicarregui et al., 1997]. For instance, the formal specification can be used as a guide for formally determining test suites. Formal specifications combined with classical (non-formal) approaches for software development has been successfully applied in many projects. Production of formal specifications helps to expose many ambiguities and omissions and yields specifications which are more concise and easier to reason about than the ones built using informal or systematic approaches [Dyer, 1992, Machado and Meira, 1995]. Real world systems can be too large to be formally proven correct in a cost-effective way, but it is fundamental to use a notation for the specification step which avoids as many anomalies as possible. Furthermore, informal specifications are not effective to uncover errors [Laski, 1988] and deriving test suites from a formal specification seems to be feasible and very promising as well as this is another way to compensate for the costs of producing it [Dick and Faivre, 1993, Parissis and Ouabdesselam, 1996, Donat, 1997]. In fact, formal specifications have been pointed out to be fundamental to the establishment of a testing theory. A possible use of formal methods and testing for verifying systems is sketched in Figure 1.1, where test obligations are generated and testing is used to discharge them along with selected proof cases [Machado, 1998, Liu, 1999].

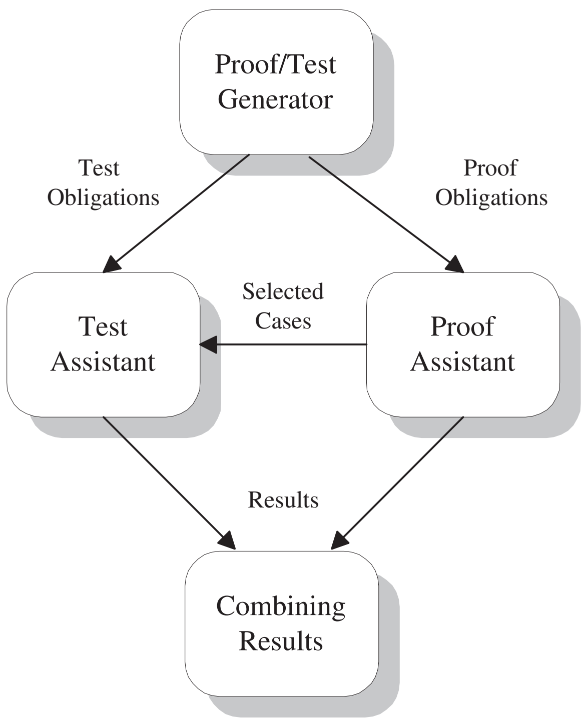

Figure 1.1: Formal verification of systems based on testing and formal proofs.

Specification-based testing, also called *formal testing*, is concerned with deriving test suites from formal specifications of programs [Richardson et al., 1989]. More recently, several works in this area have been developed focusing on:

- algebraic specifications
([Bernot, 1991], [Bernot et al., 1991], [Bernot et al., 1997], [Sankar et al., 1993], [Antoy and Gannon, 1994], [Gaudel, 1995], [Le Gall and Arnould, 1996], [Arnould et al., 1996], [Le Gall, 1999])

- model-based specifications
([Dick and Faivre, 1993], [Douglas and Kemmerer, 1994], [Stepney, 1995], [Stocks and Carrington, 1996], [Donat, 1997, Donat, 1998])

- transition systems and process algebra
([Brinksma, 1988] , [Brinksma et al., 1995], [Brinksma, 1999], [Holcombe, 1993], [Holcombe and Ipate, 1995], [Clarke and Lee, 1995], [Tabourier et al., 1999], [Díaz and Escrig, 1999], [Schneider, 1999])

- object-oriented systems
([Doong and Frankl, 1994], [Fletcher and Sajeev, 1996], [Barbey et al., 1996], [Péraire et al., 1998], [McDonald et al., 1997], [Murray et al., 1998], [Chen et al., 1998])

- reactive systems
([Richardson et al., 1992], [O'Malley et al., 1996], [Dillon and Yu, 1994], [Dillon and Ramakrishna, 1996], [Courcoubetis and Yannakakis, 1995], [Müllerburg et al., 1995], [Mandrioli et al., 1995], [Morasca et al., 1996], [Parissis and Ouabdesselam, 1996], [Jagadeesan et al., 1997])

- architecture-based systems
([Richardson and Wolf, 1996], [Bertolino and Inverardi, 1996], [Rice and Seidman, 1998], [Conquet and Marty, 1999])

The combined use of testing and model checking has also been investigated ([Ammann et al., 1998], [Halbwachs and Raymond, 1999], [Bousquet, 1999], [Holzmann and Smith, 1999], [Gargantini and Heitmeyer, 1999]). Nevertheless, a great effort is still needed in order to have testing as a standard activity in formal frameworks. For instance, the accurate interpretation of test results seems to be a crucial point. Another important point, more extensively investigated than the former, is how to properly select good finite test sets.

The main subject of this thesis is the so-called *oracle problem* which concerns the definition of decision procedures for interpreting the results of tests. Oracles are derived from structured algebraic specifications expressed in first-order logic. In this context, functionality and modularity are the main issues to be analysed.

This chapter is structured as follows. In Section 1.1, oracles are briefly introduced in the general context of testing. Section 1.2 presents some related attempts towards a formal approach to testing. Section 1.3 gives an overview of the thesis and its main contributions. Section 1.4 introduces the methodology adopted. And finally, Section 1.5 presents a plan of chapters.

1.1 Testing and Oracles

There are different types and strategies of testing which focus on different objectives and different abstract views of the software [Hamlet, 1994, Zhu et al., 1997]. The main objective is to detect errors and a good test set has a high probability of finding a hidden error. Activities like pinpointing and correcting faults are out of the scope of testing and are called *debugging*. A strategy of software testing integrates planning and test design methods and techniques in a single process. *Top-down* and *bottom-up* testing are general strategies widely used.

Authors usually differentiate between errors and faults: an *error* is an incorrect value produced by execution of the software under test and a *fault* is a bug detected in the software source code [Richardson et al., 1989]. While *dynamic testing* which involves executing the software is more likely to detect errors, *static techniques* which involves inspection or analysis of the software source code are more likely to detect faults. However, both approaches are heavily dependent on the existence of a complete and precise specification of the program [Sommerville, 1995].

The classical approaches to software testing are black-box and white-box. *Black-box* or functional or specification-based methods generate tests from the functional specification of the software while *white-box* or structural or program-based methods use information from the concrete structure and implementation of the software as a basis to derive tests. Usually, these approaches are combined, depending on the application and the objectives that should be achieved.

In general, a complete testing process includes four steps: unit testing, module testing, integration testing, and system or validation testing. *Unit testing* checks each individual component independently, *module testing* checks the relationship between all components in a module and the status of hidden and exported information, *integration testing* checks collections of modules and their combination to form sub-systems, and *system testing* checks sub-systems integration and whether the

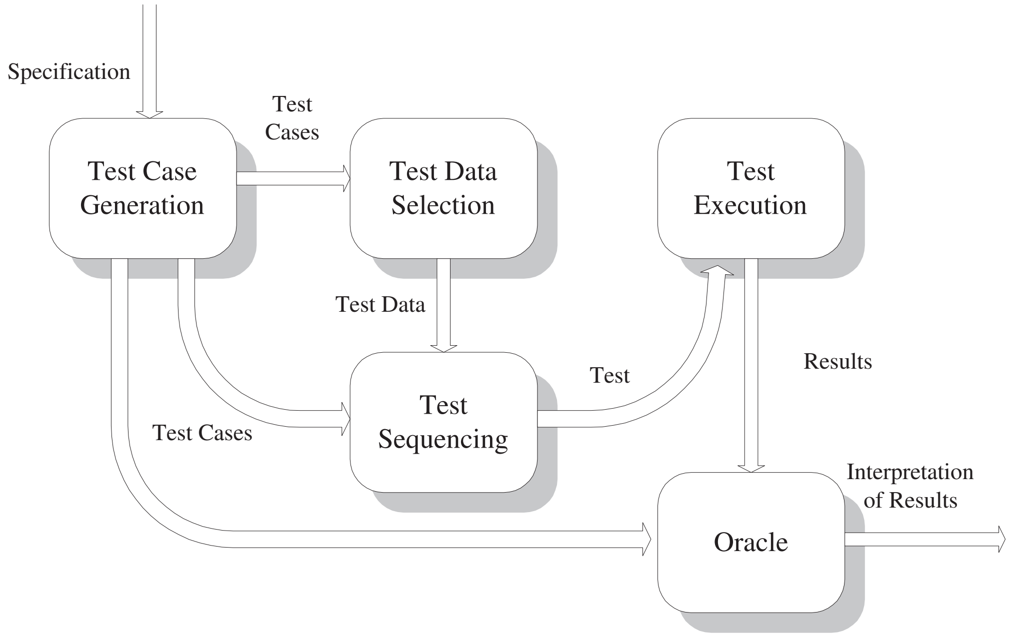

Figure 1.2: A generic model of a black-box testing process.

overall system meets its functional and non-functional requirements.

A generic model of a black-box testing process is shown in Figure 1.2. The process of designing test suites usually includes the following activities: test case generation, test data selection and test sequencing. A *test case* is a statement about what the test covers (*input criteria*, *acceptance criteria*), which can be expressed by logical predicates or even by informal statements. *Test data* or *test set* is an instance of test cases which consists of a collection of values submitted to a program as input in order to test it. *Test sequencing* produces an efficient order of submitting test data to the program avoiding repetitions [Hall and Hierons, 1991, Dick and Faivre, 1993].

Obviously, testing programs does not only consist in submitting various combinations of input values and exercising them, but also giving a coherent interpretation to the results produced by them when receiving those values. Along with selecting test data sets, specialised decision procedures, widely called *oracles*, may also be carefully planned to assist the testing process. Oracles can be classified into active and passive ones [Hoffman and Strooper, 1991]. *Passive* oracles check a static association of outputs and target outputs, whereas *active* oracles also produce the outputs to be checked. The oracle problem arises whenever such a decision procedure, which must be executable and finite, cannot be defined. For instance, this happens due to the semantic gap between specification and program values. One way of reducing this gap is by refining specifications towards automated oracles in either a given formalism or programming language [Le Gall and Arnould, 1996, McDonald et al., 1997].

Consider the following definitions of test case, test set and oracle. Given a program p, a test case may consist of a set of input values associated with expected output values, according to an axiom of the specification. Then an oracle has to check whether the output produced when the program is executed with that particular input coincides with the corresponding output in the test case. In the sequel, let p be a program whose input domain is D and output domain is X. Let TC be a test case which is defined as a total function from elements of $D' \subseteq D$ to elements of $X' \subseteq X$. Then let $dom(TC) = D'$ denotes the domain of TC. Also, let $TC(t)$ denote the corresponding target output for a given input $t \in Dom(TC)$ and $p(t)$ denote the result of executing p with input t. The program p meets TC if and only if it meets it on all inputs in $dom(TC)$, that is, $p(t) = TC(t)$ for all $t \in Dom(TC)$. However, $dom(TC)$ is likely to be infinite. Thus, a finite test set $T \subseteq dom(TC)$ needs to be selected. A function O is called an *oracle* for p on TC if for all $t \in D$ [Hamlet, 1994]:

$$O(t) = \begin{cases} true & p(t) = TC(t) \ (p(t) \text{ is an acceptable result}) \\ false & p(t) \neq TC(t) \\ true & t \notin dom(TC) \end{cases}$$

The meaning of an *acceptable result* is defined as the result of comparing $p(t)$ to the pre-defined target output $TC(t)$. The lack of an effective procedure to compare these values, mainly because $p(t)$ and $TC(t)$ are defined at different levels of abstraction, can cause the oracle problem. For instance, consider an specification of sets and an implementation of sets as lists. Suppose repetition of elements is allowed in the implementation and the following axiom needs to be checked: $S \cup T = T \cup S$. Then, let S be defined as $[1,2,2]$ and T as $[3,3,4]$, and $S \cup T = [1,2,3,3,4]$ and $T \cup S = [3,4,1,2,2]$ be the results of evaluating these expressions. Clearly, the result of the test $S \cup T$ is not literally equal to $T \cup S$, even though they are behaviourally equal. In cases like this, the oracle problem is how to define a procedure to decide that an outcome is correct according to the specification. Some approaches to sort out this problem hinges on a mapping between specification values and implementation values [Richardson et al., 1992]. Others are based on observational equivalence and observational satisfaction [Sannella and Tarlecki, 1987, Bidoit and Hennicker, 1996] of the axioms of a specification [Bernot, 1991, Gaudel, 1995, Chen et al., 1998].

The literature on specification-based testing has mainly concentrated on investigating how test cases/test sets can be refined rather than on defining oracles [Hamlet, 1994]. Some exceptions are the works of [Bernot, 1991] and [Gaudel, 1995] in the context of positive conditional specifications and [Le Gall and Arnould, 1996] which gives an interpretation by using institutions of observable results obtained from dynamic testing. Also, [Dick and Faivre, 1993] and [Richardson et al., 1992] pointed out that formal specifications can be used as oracles for software testing. Even though some theoretical issues have already been considered about oracles, a feasible method of dealing with the oracle problem has not come out yet. Furthermore, the reasons for defining automated oracles are quite obvious: human oracles are usually imprecise and error-prone while automated oracles derived from consistent and unambiguous sources can lead to more efficiency, feasibility and reliability in the testing process [Richardson et al., 1992, McDonald et al., 1997, Holzmann and Smith, 1999, Halbwachs and Raymond, 1999].

1.2 Towards a Formal Approach to Testing

Testing foundations are still weak and ill-understood. The theory of testing developed so far is still in its infancy. For instance, the fundamental problem of the semantic gap between program results and abstract specification has not been effectively solved yet. Some important attempts towards a formal approach to testing closely related to this thesis are described below.

1.2.1 A Theory of Test Selection

[Goodenough and Gerhart, 1975] present fundamental ideas about the theoretical and practical role of testing in the software development process. Their ideas have greatly influenced research in this area. They believe testing and formal proofs are complementary techniques for verification of programs and they discuss the limitation of testing as an activity without formal and well-stated procedures. They also argue that properly structured tests are able to demonstrate the absence of errors in a program. Their work is an initial attempt towards a theory of testing based on the use of a program's specification to derive test cases, illustrated by examples of problems testing must deal with. For instance, an exhaustive test set – one that exercises the program with all possible combinations of input values – is not practical. Thus, they define a *thorough* test set T, to be one which satisfies a predicate defining how some criteria C is used to select T. The criteria C describes what properties of a program must be exercised and T is complete w.r.t. C if T satisfies all properties in C. A thorough test set ensures that successful execution implies correctness with respect to C. Furthermore, they present a method for test data selection called the *condition table method* using decision table representations of program conditions and test data, where rows have condition values and columns have test case values. The approach is based on assumptions about how program errors occur and gathers information from the requirements a program is to satisfy, the program's specification and the general characteristics of the implementation method used. Also, the method defines guidelines about the sequence in which test data should be presented. The fundamental theorem of testing states that if a criteria is valid[1] and reliable[2] and a thorough (complete) test set is selected and successful, then correctness can be detected. The main difficulty is how to select thorough test sets from a given criteria and check whether the criteria is valid and reliable.

[Brinksma, 1988] presents an outline of a testing theory for the derivation of test suites in specification formalisms that allow a semantic interpretation in terms of labelled transition systems. His general ideas about conformance between program and specification have influenced the notion of correctness proposed by other works in a more general context [Le Gall and Arnould, 1996]. He also introduces the definition of a *canonical tester*, which is elsewhere called an *oracle*, for finite and non-finite systems, and sketches a proof that canonical testers always exist.

1.2.2 Testing against Algebraic Specifications

Testing from algebraic specifications has its basis in the works of [Bougé, 1985], [Bougé et al., 1986], [Bernot et al., 1991] and [Gaudel, 1995]. Test case selection and oracles to decide on discrepancies between a program behaviour and a functional specification together with modularity are the main issues investigated. Solutions to the oracle problem are proposed in [Bernot, 1991, Gaudel, 1995, Le Gall and Arnould, 1996].

[Bernot, 1991] presents a theoretical view of testing against formal specifications focusing on algebraic specifications. The main idea is that the critical properties should be proved, while less critical properties can be checked by a formal approach to testing. Using a formal specification, one can derive testing strategies in a rigorous

[1] A criteria is valid if it is possible to select data that uncovers an error.

[2] A criteria is reliable if complete test sets are consistent in the ability to reveal errors, that is, complete test sets w.r.t. this criteria are all successful or unsuccessful for a given program.

and formal framework. He introduces the idea that a test data set whose selection is based on some criteria as proposed by [Goodenough and Gerhart, 1975] cannot be considered independently of the existence of an *oracle*, a decision procedure which determines whether the program behaves correctly with respect to its specification on a given input, and of a set of hypotheses on the program which express the gap between the success of the test and the correctness of the program. The theory is based on the notion of *testing context* which is a triple (H, T, O), where T is a test data set, H is a set of hypotheses and O is an oracle. Test sets are instances of axioms and oracles are partial predicates which are either undecidable or decide whether a test is successful or not. Specifications are restricted to positive conditional equations and the theory of test data set selection explores the definition of an exhaustive test set, and a canonical test set which can be refined toward a smaller but valid[3] and unbiased[4] test set by using general data selection techniques like the regularity hypothesis and uniformity hypothesis.

As the results produced by the program may depend on some representation choices, the comparison between program and specification values gives rise to the *oracle problem*. Some approaches to handle the oracle problem for tests based on conditional positive equations are introduced. The program under test can either be instrumented with new procedures to compute abstract equalities or only observable elementary tests can be considered. Also, the oracle can be constructed with additional equality operations to drive the program under test. [Bernot et al., 1997] also defines a theory of probabilistic functional testing in which test cases are selected according to a distribution on a subdomain.

[Doong and Frankl, 1994] propose an approach to unit testing of object-oriented programs against algebraic specifications. There, a test case is defined as a pair of sequences of messages along with a tag to indicate whether these sequences should result in objects which are equivalent, that is, in the same "abstract state". Two objects of a class C are observationally equivalent if and only if either C is built-in and the objects have identical values or C is a user-defined class, and for any sequence of operations of C ending in a function returning an object of another class C', the resulting objects are observationally equivalent. Since testing cannot rely on infinite procedures, they proposed that an approximation to the observational equality should be supplied in any class. This approximation can be defined either from the abstract specification or at implementation level. [Fletcher and Sajeev, 1996] present a

[3]If testing is successful for all values in the test set, then the program is correct.

[4]Correct programs cannot be rejected by testing using the test set.

framework for semi-automatic generation of test cases for object-oriented programs from Object-Z [Duke et al., 1995] specifications based on the ideas of [Doong and Frankl, 1994]. Also, [Barbey et al., 1996, Péraire, 1998] propose an adaptation of the theory presented in [Bernot et al., 1991] to test object-oriented software taking into account the possible and impossible sequences of message call.

[Gaudel, 1995] presents a survey of research in algebraic specification-based testing. The theory presented is an extension of the one proposed by [Bernot, 1991] and is also based on the works of [Goodenough and Gerhart, 1975] and [Gourlay, 1983]. The oracle problem is more deeply analysed using the notion of observational equivalence [Bidoit et al., 1995]. A brief description of some case studies is also presented. Testing a program P against a ground equation $t = t'$ consists in evaluating the terms t and t', which yields tp and tp' respectively, and comparing the resulting values. An exhaustive test set is defined to be the set of all well-sorted ground instances of axioms. A solution to the oracle problem is proposed in the context of positive conditional specifications in which all the equations in preconditions are of observable sorts and equality on non-observable sorts is defined from a subset of all observable contexts.

[Le Gall and Arnould, 1996, Le Gall, 1999] present a formal approach to testing based on the work of [Bernot, 1991] and [Gaudel, 1995]. However, they are more concerned with the question of how far a program can be considered correct. They propose an oracle framework based on the concept of institution [Goguen and Burstall, 1984, Goguen and Burstall, 1992] and state some general definitions about correctness of a program P with respect to a specification SP via an oracle O, afterwards specialising the definitions to positive conditional equations. Institutions are used to give the semantic interpretation of specifications, programs and oracles. The definition of correctness is based on the intuition that the interpretation of observable sentences by an oracle can give semantics to programs. Test sets are defined as observable consequences of a specification SP. The aim is to avoid rejecting correct programs and detect as many incorrect programs as possible. Reference test sets T which are subsets of the set of observable consequences are constructed using equivalence partitions based on either deterministic or probabilistic choices. A program P is correct w.r.t. SP if and only if $Mod_O(P) \subseteq Mod_O(SP)$, where $Mod_O(X)$ gives the set of models of X w.r.t. O. In order to avoid computing equality on values of non-observable sorts, only observable computations are considered. Whenever SP does not have enough observations, it is made "testable" by adding the necessary ones which may require intuition and also add unnecessary complexity to the abstract specification. Due to this fact, they only consider concrete specifications refined from abstract ones. Again, the set of all observable consequences is likely to be infinite. An initial attempt towards a modular approach to testing for hierarchical specifications is also given in [Le Gall and Arnould, 1996]. [Arnould et al., 1996] also proposed to enrich algebraic specifications to allow a description of data type boundaries.

[Doche and Wiels, 2000] propose an approach, based on the works of [Gaudel, 1995] and [Péraire et al., 1998], to test generation of structured programs whose structure reflects the structure of specifications. The intention is to allow for an incremental testing approach. Like [Le Gall and Arnould, 1996], the concept of institution [Goguen and Burstall, 1984, Goguen and Burstall, 1992] is extended to include a notion of test. Tests are generated from small specifications, then they are renamed and composed according to a composition mechanism based on pushouts and preservation of correctness.

1.3 Overview of the Thesis

This thesis seeks a solution to the oracle problem for testing programs, modelled as algebras, against structured algebraic specifications with testing interface which are ordinary structured specifications extended to include test sets. Axioms are first-order formulas with the usual connectives and both universal and existential quantifiers. Structured specifications are defined using specification-building operations like *union*, *translate* and *hide* [Sannella and Tarlecki, 1997, Hennicker, 1997]. The main question addressed is: under which conditions is it possible to define a finite and executable oracle to interpret test results in this context ? In other words, what obstacles are encountered when testing from structured algebraic specifications and how can they be properly handled ? Is there a feasible solution to the oracle problem in this context ? If yes, what can be concluded about the correctness of the program being tested if testing is successful ? Behavioural satisfaction of specifications axioms by algebras is chosen to define correctness [Bidoit and Hennicker, 1996].

As [Gaudel, 1995] points out, the oracle problem often reduces to the more general problem of comparing values of a non-observable sort which makes oracles undecidable in general. However, the use of universal and existential quantifiers in specifications can make the oracle problem more difficult than in the context of positive conditional specifications which have been investigated so far [Bernot, 1991, Gaudel, 1995, Le Gall and Arnould, 1996], as infinite test sets may be required. Moreover, the structure of specifications can impose additional barriers which restrict the way specifications and test suites are defined. For instance, the structure has to be taken into account in interpreting test results w.r.t. specification axioms. Also, in order to check axioms with hidden symbols, it is necessary to provide an additional implementation for these symbols as the program under test is not supposed to implement them. Providing such an implementation may be very tricky, but essential if these axioms have to be verified.

The main contributions and outcomes of the thesis are:

- A methodical study of testing in the context of flat algebraic specifications expressed in first-order logic, focusing on the oracle problem.
- A solution to the oracle problem for flat specifications. This is based on tackling the equality problem by the use of two *approximate* equalities, one black-box and one white-box, which are applied according to specific contexts in which equations occur within axioms and taking quantifiers into account.
- A through investigation of the oracle problem by defining and comparing two extreme approaches to testing programs against structured specifications: one compositional, namely structured testing, and the other non-compositional, namely flat testing. Structured specifications are extended to include a test interface which consists of test set definitions.
- A use of specification normalisation to define a more feasible and practical solution to the oracle problem for testing from structured specifications. Three normal forms are proposed giving rise to combined styles of testing.

The thesis is centred on unit and module testing and active oracles. It does not consider refinement of test cases and test sets or deciding which degrees of confidence regarding correctness can be achieved. However, these topics are mentioned throughout this thesis insofar as it is quite unreasonable to investigate oracles without alluding to them. Also, the thesis is not aimed at giving a practical method of testing which falls beyond the intended scope. But, the importance of defining test suites from formal specifications is emphasised together with some guidelines for defining oracles. Also, rather than staying in the pure field of specification-based testing, the thesis also considers "white-box" techniques to deal with the equality problem.

1.4 Methodology

The ultimate goal of testing from algebraic specifications is to check whether specification axioms are satisfied by programs. Thus, oracles are usually active procedures which drive the necessary tests and interpret the results according to a given axiom which needs to be checked. In this thesis, oracles are defined according to a testing satisfaction relation which differs from the standard one by the way equality is computed and also because quantifiers range over test sets. For the sake of simplicity, test cases are taken as specification axioms. Obviously, techniques can always be applied in order to simplify test cases or make them more practical as in [Donat, 1997, Le Gall, 1999]. However, this out of the scope of this thesis.

The generic testing model to be followed is shown in Figure 1.3. Test cases are extracted from specifications together with test sets which are also defined at specification level. Then, oracles are defined for each test case or group of test cases. These basically consist of predicates to evaluate test cases according to test results together with procedures to compute equality on non-observable sorts (testing satisfaction). A test corresponds to the execution of a single function which is referred to in the test case. A test obligation corresponds to the combination of a test case, test data, and a test oracle. Whenever there is no chance of confusion, this is referred to simply as a test. Examples of tests and oracles are given throughout this thesis.

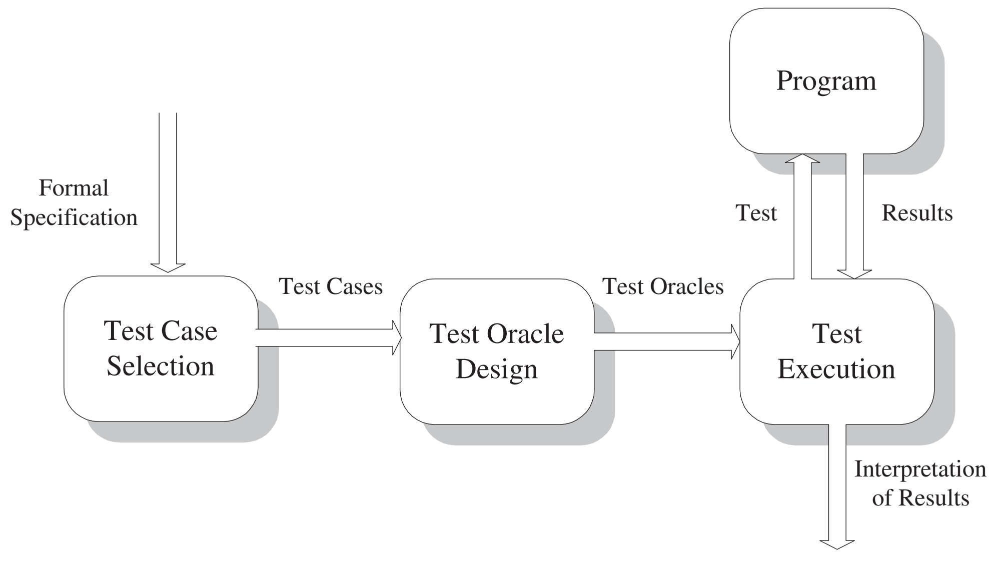

Figure 1.3: Formal testing process.

Once oracles are defined, it is necessary to check whether success (failure) in testing detected by an oracle means correctness (incorrectness). In this thesis, correctness corresponds to behavioural satisfaction of specification axioms [Bidoit and Hennicker, 1996]. But the structure of specifications also needs to be taken into account. Nevertheless, only flat specifications are considered initially in order to promote a better and progressive understanding of the oracle problem.

1.5 Outline of the Thesis

The thesis is structured as follows. Chapter 2 introduces the terminology adopted and presents some preliminary issues regarding testing from flat algebraic specifications, with a definition of oracles and test sets based on the notion of behavioural satisfaction. A correctness theorem is also introduced as a variation of the fundamental theorem of testing given by [Goodenough and Gerhart, 1975].

Chapters 3, 4 and 5 are the kernel of this thesis. Chapter 3 introduces the foundational results of the thesis which consists of a solution to the oracle problem for flat specifications where values of non-observable sorts are compared by one of two *approximate* equalities, according to the context in which the equation occurs within the axiom. The interpretation of the oracle does not make assumptions on test sets and depends on how these equalities approximate the behavioural equality and also on the use of quantifiers.

Chapter 4 deals with the oracle problem for testing from structured algebraic specifications. Specification-building operations introduce additional complications to this problem. For instance, an implementation of hidden sorts is required to check axioms composed of hidden and visible symbols. Also, translations of signatures do not always respect testing satisfaction. Furthermore, union of specifications may require test sets to coincide. A framework for testing in this context and a thorough discussion of the problems which can arise together with proposed solutions is presented.

Chapter 5 investigates the use of normalisation for testing from structured specifications. The intention is to compute a related, possibly simpler, specification, namely the *normal form*, and use it to test whenever testing from the original specification is too complex or the oracle problem arises. Three normal forms are presented.

Finally, Chapter 6 gives some conclusions and directions for future research.

Chapter 2

Preliminary Definitions

"We know less about the theory of testing, which we do often, than about the theory of program proving, which we do seldom."
J. B. Goodenough and S. L. Gerhart

Contents

This chapter introduces general terminology and preliminary concepts and ideas concerning test and oracles for verifying programs against algebraic specifications expressed in first-order logic. The oracle problem is generally surveyed in this context. As a usual assumption on algebraic specification, programs are modelled as algebras. Then oracles are defined based on a notion of behavioural satisfaction of specification axioms [Bidoit and Hennicker, 1996] with test sets defined at specification level. A concrete example is given to motivate the definitions. Finally, a general theorem is presented to show under which circumstances successful testing means correctness.

2.1 Terminology

Throughout this thesis, the reader is assumed to be familiar with general concepts of algebraic specifications [Wirsing, 1990, Astesiano et al., 1999]. The languages CASL [Language Design Group, 2000] and Standard ML (SML) [Paulson, 1996] are used in the examples, but the notation should be comprehensible without knowledge of these languages.

In the next subsections, basic concepts and notations of algebraic specification are briefly summarised along with a quick survey on CASL and SML.

2.1.1 Algebraic Concepts

Let $\Sigma = (S, F)$ be a signature with $sorts(\Sigma) = S$ and $opns(\Sigma) = F$ and let $T_\Sigma(X)$ be the Σ-term algebra, where X is an S-indexed set of countably infinite sets of variables. For any two Σ-terms t and t' of the same sort, $t = t'$ is a Σ-equation and first-order Σ-formulas are built from Σ-equations, logical connectives $(\neg,\wedge,\vee,\Rightarrow,\Leftrightarrow)$ and quantifiers $(\forall,\exists)$. A Σ-formula without free variables is called a Σ-sentence.

A Σ-algebra A consists of an S-sorted set $|A|$, the carrier sets, and for each $f : s_1 \times \ldots \times s_n \to s \in \Sigma$, an operation $(f : s_1 \times \ldots \times s_n \to s)_A : |A|_{s_1} \times \ldots \times |A|_{s_n} \to |A|_s$. We restrict to algebras with no empty carriers.

For any Σ-algebra A and valuation $\alpha : X \to |A|$, there exists a unique Σ-homomorphism $\alpha^\# : T_\Sigma(X) \to A$ which extends α. The value of $t \in |T_\Sigma(X)|_s$ in A under α is $\alpha^\#(t) \in |A|_s$, where $s \in S$. If $t \in T_\Sigma$, that is, t is a ground Σ-term, the value of t in A is $^\#(t)$, where $^\# : T_\Sigma \to A$ is the unique homomorphism.

A Σ-algebra A is reachable if and only if for every value $a \in |A|$, there exists a term $t \in T_\Sigma$ so that $^\#(t) = a$. Let B be a Σ-algebra. Then B is a subalgebra of A if $|B| \subseteq |A|$ and $f_B(b1, \ldots, b_n) = f_A(b_1, \ldots, b_n)$ for any $f : s_1 \times \ldots \times s_n \to s \in \Sigma$ and $b_1 \in |B|_{s_1}, \ldots, b_n \in |B|_{s_n}$.

Let $\sigma : \Sigma' \to \Sigma$ be a signature morphism. This extends to translate Σ'-terms to Σ-terms and Σ'-formulas to Σ-formulas. The reduct of a Σ-algebra A by σ is written $A|_\sigma$. If $\sigma : \Sigma' \hookrightarrow \Sigma$ is an inclusion, then $A|_{\Sigma'}$ may be used instead. Consider the following pushout in the category of signatures:

$$\begin{tikzcd}
& \Sigma' & \\
\Sigma_1 \arrow[ur, "\sigma'_1"] & & \Sigma_2 \arrow[ul, "\sigma'_2"'] \\
& \Sigma \arrow[ul, "\sigma_1"] \arrow[ur, "\sigma_2"'] &
\end{tikzcd}$$

For any Σ_1-algebra A_1 and Σ_2-algebra A_2 such that $A_1|_{\sigma_1} = A_2|_{\sigma_2}$, there exists a unique Σ'-algebra A' such that $A'|_{\sigma'_1} = A_1$ and $A'|_{\sigma'_2} = A_2$ (Amalgamation Lemma).

Let $SP = \langle \Sigma, \Phi \rangle$ be a flat specification, where Φ is a set of Σ-sentences, called the axioms of SP. Then $Sig(SP) = \Sigma$ denotes the signature of SP, $Alg(\Sigma)$ the class of all Σ-algebras and $Mod(SP)$ the class of Σ-algebras which satisfies the axioms of SP, that is, $Mod(SP) = \{A \in Alg(\Sigma) \mid A \models \phi \text{ for all } \phi \in \Phi\}$.

2.1.2 CASL

CASL [Language Design Group, 2000, Semantics Group, 1999] is an algebraic specification language designed by CoFI, the Common Framework Initiative for algebraic specification and development of software. This is intended as a common language for formal specification of functional requirements and modular software design. The main features include many-sorted basic specifications denoting classes of many-sorted partial first-order algebras where functions are partial or total and predicates are allowed. First-order axioms are built from equations and definedness assertions. Subsorted basic specifications are also provided. Structured specifications can be built by translation, reduction, union, extension of specifications and also generic specification with instantiation involving parameter-fitting translations also known as views. Architectural specifications and specification libraries are also part of the language.

In this thesis, CASL is used to conveniently present flat and structured specifications in the examples. For the sake of simplicity, only a subset of the language is considered by excluding features like subtyping, generic and architectural specifications which are out of the scope of the thesis. For the theoretical development, a simple subset of ASL [Wirsing, 1990, Sannella and Tarlecki, 1997] will be used (see Section 4.4). The correspondence between CASL and this subset is obvious, although spelling it out would be tedious.

2.1.3 SML

Standard ML (SML) [Paulson, 1996, Milner et al., 1997] is an applicative (functional) programming language which consists of a core language for small-scale programming and a module language for large-scale programming. The core language is a higher-order procedural language with an applicative subset and a sophisticated, strong and safe type system. The module system provides powerful and compatible facilities for building large software systems. Coherent collections of definitions can be grouped in modules called *structures*. Parameterised structures which take other structures as argument are called *functors*. Finally, *signatures* are defined in order to hide some declarations which are interesting only for implementing the major definitions.

Throughout this thesis, SML is used to illustrate oracle definitions from CASL specifications. In this case, we consider structures as algebras and boolean-valued functions as predicates. Since oracles are decision procedures, it is interesting to illustrate how can they be defined at programming level and SML seems to be a natural choice here.

2.2 Test Oracles: An Example in SML

In this section, automated oracles are defined in SML for testing SML programs against CASL specifications based on satisfaction of axioms.

Different definitions of oracles and test sets can be found in the literature on specification-based testing, depending on particular notations and formalisms [Richardson et al., 1992, Gaudel, 1995, Murray et al., 1998]. However, it is largely accepted that formal specifications, in particular specification axioms, are fundamental to the design of tests for program verification [Sommerville, 1995].

Oracles are also usually defined according to the test procedure considered. For instance, in standard testing, oracles usually receive three pieces of information to interpret test results: 1) inputs to the program, 2) outputs produced by the program and 3) target outputs that should have been produced. Item 1) is necessary if oracles are used to drive the test, which in this case corresponds to running the program and checking whether the outputs are "acceptable" according to the target outputs.

When testing programs against formal specifications, tests and oracles must be precisely planned from them in order to reach similar conclusions to the ones obtained by formal proofs regarding correctness or even partial correctness [Bernot, 1991].

Therefore, the general testing procedure may be different from the standard one. Rather than executing operations with test sets and checking whether the outputs produced match the target outputs predefined, this involves checking whether specifications axioms hold in programs for given test sets [Gaudel, 1995, Le Gall and Arnould, 1996, Machado, 1999]. In this case, target outputs are implicitly encoded in the axioms and there may be many acceptable target outputs for a given input. Oracles are predicates which evaluate axioms according to test results and tests exercise the required individual functions in the program. This is based on the standard notion of satisfaction of Σ-formulas [Wirsing, 1990]. Example 2.1 below illustrates this idea. Throughout this thesis, oracles defined in SML are also test drivers, that is, they are responsible for conducting the necessary tests.

***Example* 2.1 (Stack Oracle)** Consider the following STACK specification in CASL with the usual sorts, operations and axioms. The elements in the stack are integers.

spec STACK =
  INT
then
  sort *Stack*;
  ops $empty : Stack;$
  $push : Int \times Stack \to Stack;$
  $pop : Stack \to ?\, Stack;$
  $top : Stack \to ?\, Int;$
  pred $is\_empty : Stack;$
  vars $n : Int;$
  $s : Stack$
  - $top(push(n,s)) = n$
  - $pop(push(n,s)) = s$
  - $is\_empty(empty)$
  - $\neg\, is\_empty(push(n,s))$

end

An oracle for the STACK specification can be defined in SML as a function, `stack_o`, which computes the conjunction of its axioms according to a test set and an implementation of STACK [Mikk, 1995, McDonald et al., 1997]. The `stack_o` function is defined inside a functor, `STACK_O`, in order to allow different implementations (structures) of STACK to be tested using the same oracle. The test set consists of a list of stacks and a list of integers which are received as argument by the oracle function.

```
functor STACK_O (S : STACK) :
  sig
  (* oracle *)
   val stack_o : S.Stack list * int list -> bool;
  (* equality on Stack *)
   val == : S.Stack * S.Stack -> bool;
  end =
struct
  open S;

  infix ==;
  fun s == s' =
   if is_empty(s) andalso is_empty(s') then true
   else if is_empty(s) orelse is_empty(s') then false
        else top(s) = top(s') andalso pop(s) == pop(s');

  fun stack_o (ls,ln) =
    o_forall ls (fn s => o_forall ln (fn n =>
        (top(push(n,s)) = n) andalso
        (pop(push(n,s)) == s) andalso
        (is_empty(empty) = true) andalso
        (not (is_empty(push(n,s)) = true))));
end;
```

Notice that an explicit definition of equality on *Stack* is required in order to make the oracle function executable. This is given by "==" in the functor `STACK_O`. The reason is that *Stack* is a non-observable sort, that is, it is not identified with any particular concrete representation. Besides, it is not appropriate to simply assume that equality on values of *Stack* is the usual literal one. On the other hand, *Int* is taken as an observable sort corresponding to the pre-defined type `int` in SML whose equality is assumed to be correctly implemented. Finally, the function `o_forall` receives a list of values $[a_1, a_2 \ldots, a_n]$, a boolean function $\Psi :$ `'a -> bool` and implements the universal quantifier as a generalised conjunction, $\Psi(a_1) \wedge \Psi(a_2) \wedge \ldots \wedge \Psi(a_n)$, with one term for every possible value of the bound variable.

```
fun o_forall [] pred      = true |
    o_forall (x::xs) pred =
       (pred x) andalso o_forall xs pred;
```

The existential quantifier can also be easily defined in SML following the same idea as a generalised disjunction.

```
fun o_exists [] pred      = false |
    o_exists (x::xs) pred =
       (pred x) orelse o_exists xs pred;
```

□

The oracle defined in Example 2.1 relies on the fact that in order to formally verify a program, axioms are checked instead of individual functions. Obviously, one could also have defined an oracle function for each axiom. Unlike automated proofs, automated oracles check axioms by running implementations of functions for a given set of values. This make it possible to perform environment dependent checking of programs, where not only the code is checked, but also its execution in a given environment which may not be completely dependable. Despite the oracle problem, oracles can be conceptually simpler and easier to build than proofs.

In Example 2.1, the equality function "==" is not defined in the STACK specification. This is an instance of the *equality problem* which brings about the *oracle problem*: in other to make `stack_o` a well-formed and executable SML function, it is necessary to define "==" in `STACK_O`, but such a procedure cannot always be properly given. Moreover, due to the $\forall$ quantifier, success in testing means correctness only if the test set chosen is equivalent to the exhaustive (in this case infinite) test set. This is named here the *quantifier problem*.

Finally, it is easy to check that `top(empty)` and `pop(empty)` will never be considered, since nothing is said about them in the STACK specification. Thus, one may choose to test also from more concrete versions of STACK in order to cover these test cases [Machado, 1998].

2.3 Behavioural Equality

Equality on values of a Σ-algebra A can be interpreted by an indistinguishability relation – *behavioural equality* – a partial Σ-congruence $\approx_A$, where $\approx_A$ is denoted by a family $\approx_A = (\approx_{A,s})_{s \in S}$ of partial equivalence relations – symmetric and transitive relations – which are compatible with Σ, that is, $\forall f : s_1 \ldots s_n \longrightarrow s \in F$, $\forall a_i, b_i \in A_{s_i}$, if $a_i \approx_{A,s_i} b_i$ for all $1 \leq i \leq n$, then $f_A(a_1, \ldots, a_n) \approx_{A,s} f_A(b_1, \ldots, b_n)$. A behavioural equality $\approx_A$ is *total* if all relations are also reflexive. The definition domain of $\approx_A$ is given as the algebra of values in the relation: $Dom(\approx_A) = \{a \mid a \approx_A a\}$.

Let $Obs \subseteq S$ be a distinguished set of observable sorts. The partial observational equality $\approx_{Obs,A} = (\approx_{Obs,A,s})_{s \in S}$ is a special case of the partial behavioural equality where related elements are those which cannot be distinguished by observable computations. Let C_{Obs} be the set of all Σ-contexts $T_\Sigma(X \cup \{z_s\})$ of observable sorts with context variable z_s of sort s. Then values a and b of a non-observable sort s are observationally equal, $a \approx_{Obs,A} b$, if and only if $a, b \in {}^{\#}(T_\Sigma)$ and $\forall C \in C_{Obs} \cdot \forall \alpha : X \to {}^{\#}(T_\Sigma) \cdot \alpha_a^{\#}(C) = \alpha_b^{\#}(C)$, where ${}^{\#}(T_\Sigma)$ is the smallest subalgebra of A generated by Σ, $\alpha_a, \alpha_b : X \cup \{z_s\} \to {}^{\#}(T_\Sigma)$ are the unique extensions of α defined by $\alpha_a(z_s) = a$ and $\alpha_b(z_s) = b$. Clearly, $Dom(\approx_{Obs,A}) = {}^{\#}(T_\Sigma)$. This is the case where the set of input sorts In is empty in [Bidoit and Hennicker, 1996] which is enough for the purposes of this thesis. If A is reachable , then $\approx_{Obs,A}$ is a total observational equality.

Let $\mathcal{C} \subseteq C_{Obs}$ be an arbitrary set of observable contexts. A *contextual equality* $\sim_{\mathcal{C},A}$ is defined as follows. Values a and b are contextually equal w.r.t $\mathcal{C}$ if and only if $a, b \in {}^{\#}(T_\Sigma)$ and $\forall C \in \mathcal{C} \cdot \forall \alpha : X \to {}^{\#}(T_\Sigma) \cdot \alpha_a^{\#}(C) = \alpha_b^{\#}(C)$. Obviously, if $\mathcal{C} = C_{Obs}$, then $\sim_{\mathcal{C},A} = \approx_{Obs,A}$. Even though $\sim_{\mathcal{C},A}$ is a family of partial equivalence relations, it is not necessarily a partial congruence. But, the definition domain $Dom(\sim_{\mathcal{C},A}) = \{a \in A \mid a \sim_{\mathcal{C},A} a\}$ coincides with ${}^{\#}(T_\Sigma)$ as well.

A Σ-*behavioural equality* is a family $\approx = (\approx_A)_{A \in Alg(\Sigma)}$ of behavioural equalities, one for each Σ-algebra A. Likewise, $\approx_{Obs} = (\approx_{Obs,A})_{A \in Alg(\Sigma)}$ is a Σ-*observational equality* and $\sim_{\mathcal{C}} = (\sim_{\mathcal{C},A})_{A \in Alg(\Sigma)}$ is a Σ-*contextual equality* [Bidoit and Hennicker, 1996]. When A is obvious, the subscript of $\approx_A$ will be omitted for the sake of simplicity. Since the relations in $\approx$ are compatible with Σ, they are reflexive on values of ground terms, that is, ${}^{\#}(T_\Sigma) \subseteq Dom(\approx_A)$.

Lemma 2.2 *Let $t \in T_\Sigma(X)$ be a Σ-term and $\alpha : X \to Dom(\approx_A)$ be a valuation. Then $\alpha^{\#}(t) \in Dom(\approx_A)$.*

Proof. We need to show that $\alpha^{\#}(t) \approx_A \alpha^{\#}(t)$. By induction on the structure of t. Let $t \equiv f(t_1, \ldots, t_n) : s_1 \times \ldots \times s_n \to s$.[1] By induction hypothesis, $\alpha^{\#}(t_1) \ldots \alpha^{\#}(t_n) \in Dom(\approx_A)$. Thus, as $\approx$ is compatible with Σ, $\alpha^{\#}(f(t_1, \ldots, t_n)) \approx_A \alpha^{\#}(f(t_1, \ldots, t_n))$. □

We assume that programs, modelled as algebras, are equipped with a behavioural equality so that equality is interpreted according to this behavioural equality rather

[1]The base case is when there are no subterms, that is, n = 0.

than by the standard set-theoretical equality. For the sake of simplicity and due to the definition of the partial observational equality given in this section, where ${}^{\#}(T_\Sigma) = Dom(\approx_{Obs,A})$, we assume that ${}^{\#}(T_\Sigma) = Dom(\approx_A)$ in the general case. Partial congruences are considered insofar as testing is heavily dependent on the existence of executable programs which do have junk values. For example, suppose sets are represented as lists, but only lists without repetition are reachable. Then, lists with repetitions are never used. Therefore, it is better not restrict to total congruences and thereby reachable algebras.

Behavioural satisfaction of a Σ-formula by a Σ-algebra for a given family $\approx$ is defined as follows, where $\approx$ is used to interpret equality and valuations have their range in $Dom(\approx)$. This is the notion of correctness considered throughout this thesis.

Definition 2.3 (Behavioural Satisfaction) *Let Σ be a signature and $\approx$ be a Σ-behavioural equality. Let A be a Σ-algebra and $\alpha : X \to Dom(\approx_A)$ be a valuation. The behavioural satisfaction relation denoted by $\models_\approx$ is defined as follows.*

1. *$A, \alpha \models_\approx t = t'$ if and only if $\alpha^{\#}(t) \approx_A \alpha^{\#}(t')$;*
2. *$A, \alpha \models_\approx \neg\psi$ if and only if $A, \alpha \models_\approx \psi$ does not hold;*
3. *$A, \alpha \models_\approx \psi_1 \wedge \psi_2$ if and only if both $A, \alpha \models_\approx \psi_1$ and $A, \alpha \models_\approx \psi_2$ hold;*
4. *$A, \alpha \models_\approx \forall x : s \cdot \psi$ if and only if $A, \alpha[x \mapsto v] \models_\approx \psi$ holds for all $v \in Dom(\approx_A)$.*

where $\alpha[x \mapsto v]$ denotes the valuation α superseded at x by v.

For the sake of simplicity, the connectives $\vee, \Rightarrow, \Leftrightarrow$ and the existential quantifier $\exists$ are left out of the definition above. But they can always be defined in terms of the ones presented there in the usual way, for example, $\psi_1 \vee \psi_2 = \neg(\neg\psi_1 \wedge \neg\psi_2)$.

If $A, \alpha \models_\approx \phi$ for all $\alpha : X \to Dom(\approx_A)$, then $A \models_\approx \phi$. If $A, \alpha \models_\approx \phi$ for all $\phi \in \Phi$, then $A, \alpha \models_\approx \Phi$, where Φ is a set of Σ-formulas. When ϕ is a Σ-sentence, $A, \alpha \models_\approx \phi$ coincides with $A, \beta \models_\approx \phi$, for any valuations $\alpha : X \to Dom(\approx_A)$ and $\beta : X \to Dom(\approx_A)$. Therefore, $A \models_\approx \phi$ is written without α.

The class of algebras that behaviourally satisfy a specification $SP = \langle \Sigma, \Phi \rangle$ w.r.t. a Σ-behavioural equality $\approx$ is defined as $Mod_\approx(SP) = \{A \in Alg(\Sigma) \mid A \models_\approx \Phi\}$.

2.4 Axiomatising the Observational Equality

It is always possible and straightforward to find an infinite axiomatisation of an observational equality $\approx_{Obs}$ induced by a set *Obs* of observable sorts, although it is not always possible for the general case of behavioural equality.

In the sequel, we presented a method which covers general cases, where an infinite axiomatisation of the observational equality can always be replaced by a finite one, and also special cases where some intuition may be required. This method, given by [Bidoit and Hennicker, 1996], aims to find a finite axiomatisation of the observational equality to use in constructing behavioural proofs of theorems. They focus on using first order logic to "prove" the behavioural validity of first-order formulas. For the sake of simplicity, we consider only total observational equalities here, although the method covers partial observational equalities as well. Also, we omit some technical details from the original presentation.

Let $\mathcal{L}(\Sigma) \stackrel{\text{def}}{=} \Sigma \cup \{\sim_s : s\ s\}_{s \in S}$ be a signature, where $\sim_s$ is a predicate defining equality between two terms of sort s as an explicit denotation of $\approx_s$. For any Σ-algebra A, let $\mathcal{L}(A)$ be the $\mathcal{L}(\Sigma)$-algebra satisfying the property that it is the unique extension of A defined by $\mathcal{L}(A)|_\Sigma \stackrel{\text{def}}{=} A$ and for any $s \in S$, $\sim_s^{\mathcal{L}(A)} \stackrel{\text{def}}{=} \approx_{A,s}$. If ϕ is a Σ-sentence, $\mathcal{L}(\phi)$ is obtained by substituting "$\sim$" for "=" in ϕ. From [Bidoit and Hennicker, 1996], $A \models_\approx \phi$ if and only if $\mathcal{L}(A) \models \mathcal{L}(\phi)$.

Let $\Sigma = (S, F)$ and Σ_1 be two signatures with $\Sigma \subset \Sigma_1$,[2] and let $\Sigma_1^{\mathcal{L}} \stackrel{\text{def}}{=} \mathcal{L}(\Sigma) \cup \Sigma_1$. Let $\psi = (\psi_s)_{s \in S}$ be an S-sorted family of arbitrary Σ_1-formulas, and assume that each ψ_s has exactly two free variables, x_s and y_s. Let $AXL[\psi] = \bigwedge_{s \in S} \forall x_s, y_s \cdot [\psi_s \Leftrightarrow x_s \sim_s y_s]$ be a $\Sigma_1^{\mathcal{L}}$-sentence. Then, for any Σ_1-algebra A, the $\Sigma_1^{\mathcal{L}}$-algebra $\mathcal{AL}[\psi](A)$ is called the *axiomatic lifting* of A induced by ψ, where $\mathcal{AL}[\psi](A)|_{\Sigma_1} = A$ and $\mathcal{AL}[\psi](A) \models AXL[\psi]$. The family ψ is an axiomatisation of the Σ-behavioural equality if, for any Σ_1-algebra A, $\mathcal{AL}[\psi](A)|_{\mathcal{L}(\Sigma)} = \mathcal{L}(A|_\Sigma)$. Then $A \models_\approx \phi$ if and only if $\mathcal{AL}[\psi](A) \models \mathcal{L}(\phi)$.

Let $SP = \langle \Sigma, \Phi \rangle$ be a specification. The method given below leads to a finite axiomatisation of the Σ-observational equality for all non-observable sorts of *SP*.

1. Define $\mathcal{L}(\Sigma)$, which includes a $\sim_s$ function for each non-observable sort s and substitutes $\sim_s$ for the standard equality in all axioms.

2. Select an arbitrary set $\mathcal{C}$ of observable contexts from Σ where $\mathcal{C} \subseteq C_{Obs}$ such

[2] A larger signature Σ_1 is considered to define axiomatisations of the Σ-behavioural equality with hidden part.

that for any $s \in Obs$, the trivial context $z_s \in \mathcal{C}$ is included, where C_{Obs} is the set of observable contexts.

3. If $\mathcal{C}$ is infinite, define an adequate finite axiomatisation with hidden part $(HID, HBeh)$, where:

   - HID is a specification with finite axioms plus reachability constraints[3] of the form $\langle \Sigma_H, \mathcal{R}_H, \Phi_H \rangle$, with $\Sigma \subseteq \Sigma_H$. Basically, this specification includes hidden functions and sorts which are used to express a finite set of contexts based on $\mathcal{C}$.
   - $Beh = (Beh_s(x_s, y_s))_{s \in S}$ is an S-sorted family of finite Σ_H-formulas, where x_s and y_s are the only free variables, defined as follows:

   $$Beh_s(x_s, y_s) \stackrel{\text{def}}{=} \bigwedge_{C \in \mathcal{C}_H(s)} \forall Var(C) \cdot C[x_s] = C[y_s]$$

   where $\mathcal{C}_H$ is an arbitrary subset of observable contexts built from hidden function symbols in $\Sigma_H \backslash \Sigma$ and $Var(C)$ is the set of free variables of C.

   If $\mathcal{C}$ is finite, as we are only considering total observational equalities, a hidden part is not necessary, and then HID can be chosen as $\langle \Sigma, \emptyset, \emptyset \rangle$ and Beh as a finite family of Σ-formulas Beh_s defined as follows.

   $$Beh_s(x_s, y_s) = \bigwedge_{C \in \mathcal{C}(s)} \forall Var(C) \cdot C[x_s] = C[y_s]$$

4. Construct $AXL[Beh]$.

5. At this point, we need to check whether the axiomatisation provided by Beh is in fact an axiomatisation of $\approx_{Obs}$ w.r.t. SP. There are two possible cases:

   (a) If HID is empty, check whether the axiomatisation is a congruence, that is, SP **and** $\langle \mathcal{L}(\Sigma), AXL[Beh] \rangle \models CONG_{\Sigma}^{\sim}$, where $CONG_{\Sigma}^{\sim} = \bigwedge_{f \in F} CONG_f^{\sim}$ with $f : s_1 \ldots s_n \to s \in F$, and $CONG_f^{\sim}$ is defined as follows:

[3] A reachability constraint over a signature $\Sigma = (S, F)$ is a pair $\mathcal{R} = (S_{\mathcal{R}}, F_{\mathcal{R}})$, where $S_{\mathcal{R}} \subseteq S$, $F_{\mathcal{R}} \subseteq F$ and for any function $f \in F_{\mathcal{R}}$ with arity $s_1 \ldots s_n \to s$ the sort s belongs to $S_{\mathcal{R}}$. A Σ-algebra A satisfies $\mathcal{R}$ if for any $s \in S_{\mathcal{R}}$ and any $a \in A_s$, there exists a constructor term $t \in (T_{\mathcal{R}})_s$ and a valuation $\alpha : X' \to A$ such that $I_\alpha(t) = a$, where $X' = (X'_s)_{s \in S}$ with $X'_s = X_s$ if $s \in S \backslash S_{\mathcal{R}}$ and $X'_s = \emptyset$ if $s \in S_{\mathcal{R}}$.

$$CONG_f^{\sim} \stackrel{\text{def}}{=} \forall x_1, y_1 : s_1, \ldots, x_n, y_n : s_n \cdot$$
$$(\wedge_{1 \le i \le n} x_i \sim_{s_i} y_i) \Rightarrow f(x_1, \ldots, x_n) \sim_s f(y_1, \ldots, y_n)$$

(b) If *HID* is not empty, check whether:

i. $(SP$ **and** $HID)$ **reveal** $\Sigma = SP$ (*HID* is a conservative extension of *SP*);

ii. *SP* **and** *HID* **and** $\langle \Sigma_H^{\mathcal{L}}, AXL[Beh] \rangle \models CONG_\Sigma^{\sim}$;

iii. *Beh* has to coincide with the contextual equality induced by $\mathcal{C}$.

From $CONG_\Sigma^{\sim}$, a boolean function can be devised to check whether *Beh* is a Σ-congruence.

6. Some simplifications can be applied to $CONG_\Sigma^{\sim}$:

(a) If $\mathcal{C}$ is generated by $\Sigma' = (S, F')$ with $F' \subseteq F$, $CONG_f^{\sim}$ can be left out for all $f \in F'$;

(b) If f has only observable sorts as arguments, $CONG_f^{\sim}$ can be left out;

(c) In the premises, $x_i \sim_{s_i} y_i$ can be left out for all $s_i \in Obs$. And then y_i can be replaced by x_i in the conclusion.

(d) $\sim_s$ can be replaced by $=$ in the conclusion, if $s \in Obs$.

This method is based on the following theorem (Theorem 7.7 in [Bidoit and Hennicker, 1996]).

Theorem 2.4 *Beh is a finitary axiomatization with hidden part of the Σ-observational equality $\approx_{Obs}$ with respect to SP if and only if the following conditions are satisfied:*

1. $(SP$ ***and*** $HID)$ ***reveal*** $\Sigma = SP$ *(HID is a conservative extension of SP);*

2. *For any Σ_H-algebra $A_H \in Mod(SP$ **and** $HID)$, $\mathcal{AL}[Beh](A_H) \models CONG_\Sigma$, that is, SP **and** HID **and** $\langle \Sigma_H^{\mathcal{L}}, AXL[Beh] \rangle \models CONG_\Sigma^{\sim}$;*

3. *There exists a (possibly infinite) set $\mathcal{C} \subseteq \mathcal{C}_{Obs}$ of observable Σ-contexts which contains, for any observable sort $s \in Obs$, the trivial context z_s, and such that, for any Σ_H-algebra $A_H \in Mod(SP$ **and** $HID)$, $\mathcal{AL}[Beh](A_H) = \mathcal{AL}[Beh^{\mathcal{C}}](A_H)$, where $Beh^{\mathcal{C}}$ is the axiomatisation of the contextual equality $\sim_{\mathcal{C}}$, that is, Beh has to coincide with the contextual equality induced by $\mathcal{C}$.*

In the sequel, the following comments are pertinent:

- For all Σ-sentence ϕ:

$$SP \models_{\approx} \phi^4 \text{ if and only if } (SP \textbf{ and } HID \textbf{ and } \langle \Sigma_H^{\mathcal{L}}, AXL[Beh] \rangle) \models \mathcal{L}(\phi)$$

- The choice of an adequate subset $\mathcal{C}$ of observable contexts in step 2 may require some intuition and knowledge about the equalities under consideration. In addition to the trivial contexts, it can be enough to consider "crucial" contexts of the form $f(x_1, \ldots, x_{k-1}, z_{s_k}, x_{k+1}, \ldots, x_n)$, with $f : s_1 \times \ldots \times s_{k-1} \times s_k \times s_{k+1} \times \ldots \times s_n \to s \in F$, $s \in Obs$ and $s_k \in S \backslash Obs$. Bidoit and Hennicker argue that when the crucial contexts are neither enough nor optimal, they would at least provide a good hint about what set of contexts should be selected.

- In practice, the trivial contexts can be omitted in $\mathcal{C}$ as they would not lead to any Σ-formula in *Beh*, even though from a theoretical point of view they must be included to guarantee that $\approx_{Obs,A} = \sim_{\mathcal{C},A}$.

2.5 Behavioural Oracles

Since testing demands oracles to be executable and finite procedures, behavioural satisfaction of Σ-formulas (Definition 2.3), where quantifiers ranges over $Dom(\approx_A)$, is not a suitable basis. The definition domain is an exhaustive test set, but certainly infinite and, consequently, not workable. In this section, behavioural oracles are introduced.

2.5.1 Behavioural Testing Satisfaction

In the field of specification-based testing, test sets are usually defined from specifications rather than from programs. The reason is that the ultimate goal is to verify properties stated in the specification. Moreover, this makes it possible to design tests as specifications are created. Test sets are defined here as sets of ground terms. These terms are subsequently translated into values in the algebra under test. Evidently, in practice, test sets must be finite.

Definition 2.5 (Σ-test set) *$T = \{T_s\}_{s \in S}$ is a* Σ-test set *if $T \subseteq |T_\Sigma|$, that is, $T_s \subseteq |T_\Sigma|_s$ for all $s \in S$.*

[4] $SP \models_{\approx} \phi$ if and only if $A \models_{\approx} \phi$ for all $A \in Mod(SP)$

Behavioural oracles address the equality problem by interpreting equality up to behavioural equivalence. These oracles can be formally defined in terms of the satisfaction relation given below which differs from the behavioural satisfaction relation in Definition 2.3, since quantifiers ranges over test sets in the former rather than $Dom(\approx_A)$ in the latter.

Definition 2.6 (Behavioural Testing Satisfaction) *Let Σ be a signature, T be a Σ-test set and $\approx$ be a Σ-behavioural equality. Let A be a Σ-algebra and $\alpha : X \to Dom(\approx_A)$ be a valuation. The behavioural testing satisfaction relation denoted by $\models^T_\approx$ is defined as follows.*

1. *$A, \alpha \models^T_\approx t = t'$ if and only if $\alpha^\#(t) \approx_A \alpha^\#(t')$;*

2. *$A, \alpha \models^T_\approx \neg\psi$ if and only if $A, \alpha \models^T_\approx \psi$ does not hold;*

3. *$A, \alpha \models^T_\approx \psi_1 \wedge \psi_2$ if and only if both $A, \alpha \models^T_\approx \psi_1$ and $A, \alpha \models^T_\approx \psi_2$ hold;*

4. *$A, \alpha \models^T_\approx \forall x : s \cdot \psi$ if and only if $A, \alpha[x \mapsto v] \models^T_\approx \psi$ holds for all $v \in {}^\#(T)_s$.*

where $\alpha[x \mapsto v]$ denotes the valuation α superseded at x by v.

Again, when ϕ is a Σ-sentence, $A \models^T_\approx \phi$ is written without α. If $A, \alpha \models^T_\approx \phi$ for all $\phi \in \Phi$, then $A, \alpha \models^T_\approx \Phi$, where Φ is a set of Σ-formulas.

Despite the oracle problem, tests based on behavioural oracles can be naturally integrated into formal development frameworks where formal proofs are performed for verifying some parts of the specification while others are checked by testing. If we assume "==" is interpreted up to behavioural equivalence, Example 2.1 presents a behavioural oracle which checks whether specification axioms are satisfied by an implementation of STACK and a test set, according to Definition 2.6.

2.5.2 Exhaustive, Valid and Unbiased Test Sets

Exhaustive test sets which exercise a program with all possible combinations of values are usually infinite and some refinements must be applied to make them finite. But, not all test sets are interesting, unless they are *valid* and/or *unbiased* [Bernot, 1991, Gaudel, 1995]. Unbiased test sets are ones which detect all correct programs, but incorrect programs can also be detected as correct. In other words, they do not cause correct programs to be (erroneously) rejected. On the other hand, valid test sets do not accept incorrect programs, but correct programs can be rejected. Therefore, an ideal test set, which would allow correctness/incorrectness to be precisely detected, must be valid and unbiased.

Definition 2.7 (Exhaustive, Valid and Unbiased Σ-test sets) *Let T be a Σ-test set, $\approx$ be a Σ-behavioural equality and ϕ be a Σ-formula. Let A be a Σ-algebra and $\alpha : X \to Dom(\approx_A)$ be a valuation.*

- *T_Σ is an* exhaustive Σ-test set*;*
- *T is a* valid Σ-test set *for $A, \alpha, \approx, \phi$ if and only if $A, \alpha \models^T_\approx \phi$ implies $A, \alpha \models^{T_\Sigma}_\approx \phi$;*
- *T is an* unbiased Σ-test set *for $A, \alpha, \approx, \phi$ if and only if $A, \alpha \models^{T_\Sigma}_\approx \phi$ implies $A, \alpha \models^T_\approx \phi$;*

Any $T \subseteq T_\Sigma$ is unbiased in the context of positive conditional specifications investigated in [Gaudel, 1995, Le Gall, 1999]. Nevertheless, T is not always unbiased in general. For example, if ϕ contains only existential quantifier and no negation, then $A, \alpha \models^T_\approx \phi$ implies $A, \alpha \models^{T_\Sigma}_\approx \phi$, but the converse may not hold.

Notice that the definition of valid and unbiased also depends on a valuation α because ϕ is a Σ-formula that may contains free variables. For the sake of simplicity, when ϕ is a Σ-sentence, we omit α, saying that T is valid/unbiased for $A, \approx, \phi$.

Finite valid and unbiased test sets may be difficult to define in practice. It is likely that only infinite test sets are both valid and unbiased, but these are impractical. Thus, in some cases, it may only be possible to define finite test sets which are either valid *or* unbiased. In the context of positive conditional specifications, a *complete* test set is commonly referred to as the best test set we can define which is unbiased and rejects as many incorrect programs as possible [Le Gall and Arnould, 1996, Le Gall, 1999]. In this case, test set selection is based on hypotheses which reduce the gap between success of tests and the correctness of programs [Bernot, 1991].

2.5.3 Correctness

Thus far, test sets and oracles based on behavioural satisfaction of axioms were introduced. But, what can be concluded about the correctness of programs based on successful tests ? The oracle problem in the context of algebraic specification expressed in first-order logic reduces to the equality problem, that is, how a procedure for comparing values of non-observable sorts can be defined, and the quantifier problem, that is, how quantifiers which demand exhaustive test sets can be handled.

Undoubtedly, a solution to the oracle problem depends on how well equality and quantifier problems are tackled so that oracles can decide upon correctness.

Behavioural oracles confront the equality problem by interpreting equality up to behavioural equivalence. Regarding the quantifier problem, valid and unbiased test sets are an alternative. Nevertheless, in Definition 2.7, test sets are exhaustive with respect to the algebra of ground Σ-terms, not the Σ-algebra A. In other words, test sets are exhaustive with respect to specifications not programs. Therefore, behavioural satisfaction of specifications by programs for either exhaustive or valid and unbiased test sets is only guaranteed if $Dom(\approx_A) = {}^{\#}(T_\Sigma)$, that is, all values in the relation $\approx$ are reachable by ground terms (the definition domain $Dom(\approx_A)$ is reachable). Correspondingly, ordinary satisfaction is only guaranteed if all program values are reachable [Gaudel, 1995].

Theorem 2.8 below shows under what circumstances successful test can lead to a guarantee of correctness. As mentioned in Section 2.3, we are assuming $Dom(\approx_A) = {}^{\#}(T_\Sigma)$

Theorem 2.8 (Correctness) *Let T be a Σ-test set, $\approx$ be a Σ-behavioural equality and ϕ be a Σ-formula. Let A be a Σ-algebra and $\alpha : X \to Dom(\approx_A)$ be a valuation. If T is valid and unbiased for $A, \alpha, \approx, \phi$, then*

$$A, \alpha \models_\approx \phi \text{ if and only if } A, \alpha \models_\approx^T \phi$$

Proof. ($\Rightarrow$) Suppose $A, \alpha \models_\approx \phi$. Then, by Definitions 2.3 and 2.6 and because $Dom(\approx_A) = {}^{\#}(T_\Sigma)$, $A, \alpha \models_\approx^{T_\Sigma} \phi$. Since T is unbiased for $A, \alpha, \approx, \phi$, then $A, \alpha \models_\approx^T \phi$. ($\Leftarrow$) Suppose $A, \alpha \models_\approx^T \phi$. Since T is valid for $A, \alpha, \approx, \phi$, then $A, \alpha \models_\approx^{T_\Sigma} \phi$. Hence, $A, \alpha \models_\approx \phi$. □

Theorem 2.8 states that, whenever we have a valid and unbiased test set, testing is equivalent to behavioural satisfaction and, consequently, can be precisely used instead of proofs to verify whether a program P, modelled by A, conforms to a specification $SP = \langle \Sigma, \Phi \rangle$, that is, $A \models_\approx \phi$, for all $\phi \in \Phi$. This theorem is a variation of the fundamental theorem of testing given by [Goodenough and Gerhart, 1975], considering specification axioms as the criterion to be checked.

2.6 Concluding Remarks

Apart from basic terminology, this chapter presents automated oracles in SML defined from CASL specifications (Example 2.1). Clearly, despite the equality problem, the basic procedure for defining these oracles is straightforward no matter how complicated axioms are, whereas the complexity of proofs may grow as specifications becomes more complex.

In order to cope with the equality problem, behavioural oracles are introduced based on behavioural satisfaction of specification axioms by a program for a given test set. According to Theorem 2.8, valid and unbiased test sets are required so that the quantifier problem is handled and a conclusion of correctness can be reached when tests are successful. Nevertheless, finite valid and unbiased test sets as well as finite axiomatisations of the behavioural equality may be too difficult to define in practice, if not impossible. Therefore, behavioural oracles do not lead to a practical solution to the oracle problem. Chapter 3 in the sequel presents a more feasible solution to this problem based on approximate oracles.

Chapter 3

The Approximate Oracle

"The criterion which we use to test the genuineness of apparent statements of fact is the criterion of verifiability. We say that a sentence is factually significant to any given person, if, and only if, he knows how to verify the proposition which it purports to express – that is, if he knows what observations would lead him, under certain conditions, to accept the proposition as being true, or reject it as being false."
A. J. Ayer

Contents

This chapter presents a solution to the oracle problem for flat algebraic specifications. As [Bernot, 1991, Gaudel, 1995] points out, the oracle problem often reduces to the general problem of comparing two values of a non-observable sort (equality problem). However, when both existential and universal quantifiers are considered this makes the oracle problem more difficult than in the context of positive conditional specifications which has been investigated so far [Bernot, 1991, Gaudel, 1995, Le Gall and Arnould, 1996], since infinite test sets may be required (quantifier problem). The solution, proposed in [Machado, 1999], consists in defining approximate oracles with two approximate equalities which are applied according to the context in which equations occur. Quantifiers are taken into account so that these oracles can be defined independently of test sets.

3.1 The Equality Problem

The equality problem arises when two values of a non-observable sort need to be compared and the respective equality is not specified and implemented. This makes oracles undecidable in general. One solution to this problem is to define behavioural oracles which interpret this equality up to behavioural equivalence as discussed in the previous chapter.

In this section, two classical approaches to give an implementation of equality for testing purposes are reviewed. More precisely, these approaches aim to give a procedure for comparing values of a *non-observable* sort, since *observable* sorts are usually identified with the ones predefined in programming languages. The approaches, namely *black-box* and *white-box*, differ fundamentally on the level of abstraction at which equality is defined. The white-box approach uses the concrete representation of sorts, whereas the black-box is based on a finite axiomatisation of the observational equality.

3.1.1 White-Box Approach

The white-box approach consists in defining equality on values of a non-observable sort s by using information about the internal concrete representation of s. An obvious way of doing this is by comparing the components of this representation. However, this does not always give the intended equality, since different internal representations may correspond to a single value at abstract level.

In this approach, in order to test a Σ-algebra A against $SP = \langle \Sigma, \Phi \rangle$, one may define a version of A, namely $\mathcal{L}(A)$, which implements equalities for non-observable sorts according to the signature $\mathcal{L}(\Sigma)$ in terms of the underlying representation. An oracle O then receives $\mathcal{L}(\Sigma)$-algebras as argument instead of Σ-algebras. This is illustrated in Example 3.1, where a white-box equality is defined and used by the oracle to compare values of a non-observable sort.

***Example* 3.1 (White-Box Equality)** Consider again the STACK specification given in Example 2.1 and suppose one decides to implement the *Stack* sort in terms of *Array* which is specified as follows.

spec ARRAY =
  INT
then
  sort *Array*;
  ops $empty : Array$;
  $put : Int \times Int \times Array \to Array$;
  $retrieve : Int \times Array \to Int$;
  vars $n, m, v, w : Int$;
  $a : Array$
- $put(n, v, put(n, w, a)) = put(n, v, a)$
- $\neg\, n = m \Rightarrow put(n, v, put(m, w, a)) = put(m, w, put(n, v, a))$
- $retrieve(n, empty) = 0$
- $retrieve(n, put(n, v, a)) = v$
- $\neg\, n = m \Rightarrow retrieve(n, put(m, v, a)) = retrieve(n, a)$

end

The implementation of STACK by ARRAY is given as follows. *Stack* is defined as a pair `(Array, Int)`, where the first is an array value for representing the elements of the stack and the second is an auxiliary pointer for indicating the number of elements in the stack.

```
functor stack (A : ARRAY) : STACK =
  struct
    type Stack = A.Array * int;
    val empty = (A.empty,0);
    fun push(n,(a,i)) = (A.put(i,n,a),i+1);
    fun top(a,i) = A.retrieve(i-1,a);
    fun pop(a,i) = (a,i-1);
    fun is_empty(a,i) = (i=0);
  end;
```

Consider the signature LSTACK and the oracle functor `LSTACK_O` given below, where the former extends STACK by an equality predicate and the latter checks LSTACK implementations rather than STACK ones.

spec LSTACK =
  STACK
then
  pred $eq\ :\ Stack \times Stack$;
end

```
functor LSTACK_O (S : LSTACK) :
  sig
   (* oracle *)
   val o_S : S.Stack list * int list -> bool;
   (* white box equality on Stack *)
   val == : S.Stack * S.Stack -> bool;
  end =
struct
  open S;

  infix ==;
  fun s == s' = eq(s,s');

  fun o_S (ls,ln) =
    o_forall ls (fn s => o_forall ln (fn n =>
         (top(push(n,s)) = n) andalso
         (pop(push(n,s)) == s) andalso
         (is_empty(empty) = true) andalso
         (not (is_empty(push(n,s)) = true))));
end;
```

The following functor `lstack` is an extension of the implementation of STACK given by the `stack` functor, including an implementation of equality on values of *Stack* (`eq`). Intuitively, two stacks are equal if they have the same number of elements and each element in a given position below the pointer is equal to the element occupying the same position in the other stack and vice-versa. Even if the related array is not empty, if the pointer is zero, the stack is empty. Thus, for any stack a, $\mathtt{eq}(\mathtt{empty},\mathtt{pair}(\mathtt{a},0)) = \mathtt{true}$.

```
functor lstack (A : ARRAY) : LSTACK =
  struct
```

```
    type Stack = A.Array * int;
    .
    .
    .
    (* numbers less than an integer i *)
    fun list_int 0 = []
      | list_int i = (i-1)::list_int(i-1);

    fun eq((a,i),(a',i')) =
      let val lj = list_int(i) in
          (i=i') andalso
          (o_forall lj (fn j =>
            A.retrieve(j,a)=A.retrieve(j,a')))
      end
  end;
```

□

It can be argued that the white-box approach is just a way of adding an explicit equality operation to the algebra, and it corresponds to adding the operation "*eq*" and specifying it by axioms in *SP* and then using these axioms to derive a computable equality. In other words, one might encourage specifiers to explicitly specify and implement equality for all new sorts. Nevertheless, this is not the case and this solution would lead to overspecification in *SP* which may be inappropriate, because the equality implementation may only be necessary at testing time. Notice that specifications and implementations under test may remain unchanged while lifted versions of them are constructed in order to incorporate the explicit definitions of equality. This avoids, for instance, introducing bugs in the actual programs.

The white-box approach relies on concrete representations and most of the time on the use of intuition whereas the black-box approach introduced in the next section looks into a more automatic way of deriving equality from an abstract specification.

3.1.2 Black-Box Approach

The black-box approach relies on using information provided by the abstract formal specification in order to compare values of a non-observable sort by the observational equality. This approach was first investigated by [Bernot, 1991], [Gaudel, 1995] and [Le Gall and Arnould, 1996] in the context of positive conditional specifications.

An oracle which interprets equality up to observational equivalence should consider all observable contexts which is likely to lead to an infinite process.

[Bernot, 1991, Gaudel, 1995] argue that some heuristics called "oracle hypotheses" may be applied in order to reduce the number of observable contexts that need to be considered. In this case, equality can be erroneously found to be true. Table 3.1 below summarises possible combinations of errors originated from wrong equality interpretations in a positive conditional axiom. According to line **3**, incorrect programs are liable to be accepted, which is not a crucial mistake since testing is regarded as not suitable for proving correctness, that is, the complete absence of errors, but for showing the presence of errors [Dijkstra, 1981]. In order to avoid rejecting correct programs, [Gaudel, 1995] restricted tests to be defined from positive conditional equations where all preconditions of equations are of observable sorts, which restricts attention to lines **3** and **4**, since we assume that equality of observable sorts is directly computable.

	Premises of ϕ	**Conclusion of ϕ**	ϕ	**Consequence**
1	true (error)	false	false (error)	Reject correct programs
2	true (error)	true	true	Ok
3	true	true (error)	true (error)	Accept incorrect programs
4	false	true (error)	true	Ok

Table 3.1: Results and consequences of erroneously evaluating positive conditional axioms, when equality is erroneously found to be true.

From a more practical point of view, the main problem to be investigated in the black-box approach is how to turn the infinite set of observable contexts into a finite one so that axioms with equations of non-observable sorts in the premises can also be handled as well as formulae more complicated than positive conditional equations. Example 3.2 below illustrates the application of the method proposed by [Bidoit and Hennicker, 1996] (Section 2.4) to find such a finite axiomatisation of the observational equality.

***Example* 3.2 (Black-Box Equality)** Consider the STACK specification given in Example 2.1. In this example, the method presented in Section 2.4 is employed to define equality on *Stack*.

1. Firstly, we select the following subset of observable contexts which corresponds to the trivial and crucial ones:

$$\mathcal{C} = \{top(z_{stack}), is\_empty(z_{stack}), z_{int}, z_{bool}\}$$

Evidently, equality on stacks does not only depend upon the elements at the top of the stacks and whether the stacks are empty or not. We must also consider all other elements and their positions in the stacks. Thus, the trivial and crucial contexts are not enough to observe values of *Stack*. At this point, we need some intuition to improve the definition of $\mathcal{C}$. Clearly, we need to check the equality of the elements in the top and then apply pop and compare the elements on the top again and so on. Thus, $\mathcal{C}$ is an infinite set of observable contexts:

$$\begin{aligned}\mathcal{C} = \;& \{top(z_{stack}), top(pop(z_{stack})), top(pop(pop(z_{stack}))), \ldots\} \cup \\ & \{is\_empty(z_{stack}), z_{int}, z_{bool}\}\end{aligned}$$

2. Secondly, because $\mathcal{C}$ is infinite, we need to define a specification with a hidden part called HID with additional operations. The intention is to substitute an equivalent finite set of observable contexts, namely $\mathcal{C}_H$, for $\mathcal{C}$, by making use of auxiliary functions.

spec HID =
  STACK
then
  NAT
then
  ops *topn* : $Nat \times Stack \rightarrow Int$;
    height : $Stack \rightarrow Nat$;
  vars x : Nat;
    s : $Stack$;
    n : Int
  - $topn(0,s) \;=\; top(s) \;\wedge\; topn(suc(x),s) \;=\; topn(x,pop(s))$
  - $height(empty) \;=\; 0 \;\wedge\; height(push(n,s)) \;=\; suc(height(s))$

end

where *Nat* is the usual type of natural numbers with constructors 0 and *suc*. Then $\mathcal{C}_H$ and, consequently, *Beh* (axiomatisation of equality on *Stack*) are defined as follows.

$$\mathcal{C}_H = \{topn(x, z_{stack}), height(z_{stack}), z_{int}, z_{bool}\}$$

$$Beh_{stack}(s,s') = \begin{array}{l} height(s) = height(s') \wedge \\ \forall x : Nat \cdot topn(x,s) = topn(x,s') \end{array}$$

3. Thirdly, the corresponding $AXL[Beh]$ induced by Beh is defined as follows.

$$\forall s,s' : stack \cdot \begin{array}{l} height(s) = height(s') \wedge \\ \forall x : Nat \cdot topn(x,s) = topn(x,s') \Leftrightarrow s \sim_{stack} s' \end{array}$$

   Let STACK_O and LSTACK be as in Example 3.1. Then, from AXL, eq is defined in the following implementation of LSTACK.

```
fun list_nat zero = []
  | list_nat (suc x) =  x::list_nat(x);

functor lstack (S : HID) : LSTACK =
 struct
  open S;
  fun eq(s,s') = height(s) = height(s') andalso
                 let val lx = list_nat(height(s)) in
                     o_forall lx (fn x =>
                         topn(x,s) = topn(x,s'))
                 end;
end;
```

   where list_nat(x) returns a list of natural numbers. The intention is to limit the application of topn to the height of the stacks. Even though *Beh* is a finite axiomatisation, *Nat* is infinite and not all natural numbers are useful, except the ones which point to values on the stacks.

4. Finally, we need to check whether conditions in item 5(b) (Section 2.4) are met, that is, *Beh* is an axiomatisation of $\approx_{Obs,Stack}$. Condition (i) is straightforward by definition of HID and condition (iii) by definition of *topn* and *height*. However, to check condition (ii) we define $CONG_{\Sigma}^{\sim}$ which after the simplifications suggested in steps 6(a), 6(c) and 6(d) is as follows.

$$\forall n : int; s,s' : stack \cdot (s \sim_{stack} s') \Rightarrow \begin{array}{l} (push(n,s) \sim_{stack} push(n,s')) \wedge \\ (is\_empty(s) = is\_empty(s')) \end{array}$$

   In order to check (ii) by testing, $CONG_{\Sigma}^{\sim}$ can be turned into a boolean function and conventional testing procedures like submitting a test set consisting of different pairs of values can be applied. Nevertheless, this test set would probably be infinite and one way of reducing it is by only selecting pairs of equal values according to *Beh* since submitting values which are different gives no information. □

Unlike the white-box approach, the black-box approach presented in this section is a semi-automated method for defining equality on non-observable sorts. However, this method can be difficult to employ in practice. Intuition is still required and extra complications can be added to oracles. For instance, in Example 3.2, an additional set of data, possibly infinite, is required. In some cases, even if crucial contexts are sufficient and the axiomatisation is finite, due to the fact that these contexts may have parameters other than the context variable, computing this axiomatisation may depend on infinite sets of data as will be shown in Example 3.30 in Section 3.6. Furthermore, it is also necessary to check whether the equality implements a congruence.

3.2 Approximate Equalities

Implementing equality procedures based on behavioural equivalence may be too complicated or even impossible in practice. The white-box approach is heavily dependent on intuition, while the black-box approach, even though partially automated, can be arduous in practice. When neither the real equality nor a finite axiomatisation of the observational equality can be properly given, we are left with the option of defining approximations only. Nevertheless, if these approximations are appropriately employed, the equality problem can be solved, as we show in this chapter. But, before going into more details about this, we define what an approximate equality is.

Definition 3.3 (Approximate Equality) *Let A be a Σ-algebra. A family of binary relation $\sim_A = (\sim_{A,s})_{s \in S}$ on A is called an* approximate equality.

It is reasonable to require $\sim_A$ to be reflexive as well in Definition 3.3, even though this assumption is not necessary to the results presented in this thesis. As before, a Σ-approximate equality $\sim = (\sim_A)_{A \in Alg(\Sigma)}$ is a family of approximate equalities, one for each Σ-algebra A. The attentive reader will notice that contextual equalities $\sim_{C,A}$, introduced in Chapter 2, are approximate equalities, albeit not behavioural equalities. Obviously, behavioural equalities are approximate equalities as well.

When compared to the behavioural equality, any approximate equality can be classified as *sound* if all values that it identifies are indeed equal, or *complete* if all equal values are identified. In general, the white-box approach will produce sound equalities, whereas the black-box will produce complete equalities. For instance, any contextual equality is complete.

Definition 3.4 (Sound Equality) *Let A be a Σ-algebra and $\sim_A$ be an approximate equality on A. Then $\sim_A$ is a* sound equality *if and only if* $\forall a, a' \cdot (a, a' \in Dom(\approx_A)) \Rightarrow (a \sim_A a' \Rightarrow a \approx_A a')$.

The condition $(a, a' \in Dom(\approx_A))$ is due to the fact that $\approx_A$ is a partial congruence and values not in $Dom(\approx_A)$ need not be considered. This is not necessary in the definition of complete equality given below as $\approx_A$ is symmetric and transitive and then $a \approx_A a'$ implies $a, a' \in Dom(\approx_A)$.

Definition 3.5 (Complete Equality) *Let A be a Σ-algebra and $\sim_A$ be an approximate equality on A. Then $\sim_A$ is a* complete equality *if and only if* $\forall a, a' \cdot a \approx_A a' \Rightarrow a \sim_A a'$.

A Σ-approximate equality $\sim$ is sound (complete) w.r.t to the Σ-behavioural equality $\approx$ if for each Σ-algebra A, $\sim_A$ is sound (complete) w.r.t. $\approx_A$.

Sound or complete equalities can be successfully applied instead of the behavioural equality in specific contexts. These contexts are equality occurrences in a Σ-formula which might be either positive or negative in the following sense.

Definition 3.6 (Equality occurrences) *The set of occurrences Occ of Σ-equations in a first-order Σ-formula is defined as follows.*

1. $Occ[t = t'] \stackrel{def}{=} \{\varepsilon\}$
2. $Occ[\neg\psi] \stackrel{def}{=} \{-1 \cdot \omega \mid \omega \in Occ(\psi)\}$
3. $Occ[\psi_1 \wedge \psi_2] \stackrel{def}{=} \{1 \cdot \omega \mid \omega \in Occ(\psi_1)\} \cup \{2 \cdot \omega \mid \omega \in Occ(\psi_2)\}$
4. $Occ[\psi_1 \vee \psi_2] \stackrel{def}{=} \{1 \cdot \omega \mid \omega \in Occ(\psi_1)\} \cup \{2 \cdot \omega \mid \omega \in Occ(\psi_2)\}$
5. $Occ[\psi_1 \Rightarrow \psi_2] \stackrel{def}{=} \{-1 \cdot \omega \mid \omega \in Occ(\psi_1)\} \cup \{2 \cdot \omega \mid \omega \in Occ(\psi_2)\}$
6. $Occ[\psi_1 \Leftrightarrow \psi_2] \stackrel{def}{=} Occ(\psi_1 \Rightarrow \psi_2) \cup Occ(\psi_2 \Rightarrow \psi_1)$
7. $Occ[Qx\psi] \stackrel{def}{=} \{1 \cdot \omega \mid \omega \in Occ(\psi)\}$, *where* $Q = \{\forall, \exists\}$

$$\begin{array}{ccccccc}
eq_1 & \Rightarrow & eq_2 & & & & \\
- & & + & & & & \\
(eq_1 & \Rightarrow & eq_2) & \wedge & eq_3 & & \\
- & & + & & + & & \\
((eq_1 & \Rightarrow & eq_2) & \wedge & eq_3) & \Rightarrow & eq_4 \\
+ & & - & & - & & +
\end{array}$$

$$\begin{array}{ccccccccc}
\forall X & \cdot & ((eq_1 & \Rightarrow & eq_2) & \wedge & eq_3) & \Rightarrow & eq_4 \\
 & & + & & - & & - & & +
\end{array}$$

Figure 3.1: Positive and negative occurrences of equations.

where "$-$" expresses that the formula is in a negative position. An occurrence in $Occ[\Psi]$ *is* positive *if it has an even number of* -1*, otherwise it is* negative*.*

Notice that both left and right equality occurrences in $\Leftrightarrow$ can be positive and negative depending on which direction of the implication we are looking at.

***Example* 3.7 (Equality Occurrences)** Let eq_1, eq_2, eq_3 and eq_4 be equations. Let $\forall X \cdot ((eq_1 \Rightarrow eq_2) \wedge eq_3) \Rightarrow eq_4$ be a Σ-formula. The occurrences of these equations in this formula is computed below and illustrated in Figure 3.7.

$Occ[eq_1 \Rightarrow eq_2] = \{-1 \cdot \varepsilon, 2 \cdot \varepsilon\}$

$Occ[(eq_1 \Rightarrow eq_2) \wedge eq_3] = \{1 \cdot -1 \cdot \varepsilon, 1 \cdot 2 \cdot \varepsilon, 2 \cdot \varepsilon\}$

$Occ[((eq_1 \Rightarrow eq_2) \wedge eq_3) \Rightarrow eq_4] =$

$\{-1 \cdot \omega \mid \omega \in Occ[(eq_1 \Rightarrow eq_2) \wedge eq_3)\} \cup \{2 \cdot \omega \mid \omega \in Occ(eq_4)\} =$

$\{-1 \cdot 1 \cdot -1 \cdot \varepsilon, -1 \cdot 1 \cdot 2 \cdot \varepsilon, -1 \cdot 2 \cdot \varepsilon\} \cup \{2 \cdot \varepsilon\}$ □

When two approximate equalities are applied together – one sound and one complete, one in positive and the other in negative occurrences of equations – they are enough to sort out the equality problem under certain conditions as we show in the next section.

3.3 Testing Satisfaction

Regarding behavioural satisfaction, *approximate oracles* decide whether specification axioms are satisfied by algebras for given test sets and pairs of approximate equalities.

These oracles are defined based on the testing satisfaction relation of Σ-formulas given below. This relation is a generalisation of the standard notion of behavioural satisfaction (Definition 2.3) with equality interpreted by approximate equalities and quantifiers ranging over given test sets.

Definition 3.8 (Testing Satisfaction) *Let Σ be a signature, T be a Σ-test set and $\sim, \doteqdot$ be two Σ-approximate equalities. Let A be a Σ-algebra and $\alpha : X \to Dom(\approx_A)$ be a valuation. The testing satisfaction relation denoted by $\models^T_{\sim,\doteqdot}$ is defined as follows.*

1. *$A, \alpha \models^T_{\sim,\doteqdot} t = t'$ if and only if $\alpha^{\#}(t) \sim_A \alpha^{\#}(t')$;*
2. *$A, \alpha \models^T_{\sim,\doteqdot} \neg\psi$ if and only if $A, \alpha \models^T_{\doteqdot,\sim} \psi$ does not hold;*
3. *$A, \alpha \models^T_{\sim,\doteqdot} \psi_1 \wedge \psi_2$ if and only if both $A, \alpha \models^T_{\sim,\doteqdot} \psi_1$ and $A, \alpha \models^T_{\sim,\doteqdot} \psi_2$ hold;*
4. *$A, \alpha \models^T_{\sim,\doteqdot} \psi_1 \vee \psi_2$ if and only if either $A, \alpha \models^T_{\sim,\doteqdot} \psi_1$ or $A, \alpha \models^T_{\sim,\doteqdot} \psi_2$ holds;*
5. *$A, \alpha \models^T_{\sim,\doteqdot} \psi_1 \Rightarrow \psi_2$ if and only if $A, \alpha \models^T_{\sim,\doteqdot} \psi_2$ holds if $A, \alpha \models^T_{\doteqdot,\sim} \psi_1$ holds;*
6. *$A, \alpha \models^T_{\sim,\doteqdot} \psi_1 \Leftrightarrow \psi_2$ if and only if both $A, \alpha \models^T_{\sim,\doteqdot} \psi_1 \Rightarrow \psi_2$ and $A, \alpha \models^T_{\sim,\doteqdot} \psi_2 \Rightarrow \psi_1$ hold;*
7. *$A, \alpha \models^T_{\sim,\doteqdot} \forall x : s \cdot \psi$ if and only if $A, \alpha[x \mapsto v] \models^T_{\sim,\doteqdot} \psi$ holds for all $v \in {}^{\#}(T)_s$;*
8. *$A, \alpha \models^T_{\sim,\doteqdot} \exists x : s \cdot \psi$ if and only if there exists $v \in {}^{\#}(T)_s$ such that $A, \alpha[x \mapsto v] \models^T_{\sim,\doteqdot} \psi$ holds.*

where $\alpha[x \mapsto v]$ denotes the valuation α superseded at x by v. In this relation, $\sim$ is always applied in positive positions and $\doteqdot$ is always applied in negative positions. Note that equalities are reversed when negative positions are reached.

If $A, \alpha \models^T_{\sim,\doteqdot} \phi$ for all $\phi \in \Phi$, then $A, \alpha \models^T_{\sim,\doteqdot} \Phi$, where Φ is a set of Σ-formulas. When ϕ is a Σ-sentence, $A, \alpha \models^T_{\sim,\doteqdot} \phi$ coincides with $A, \beta \models^T_{\sim,\doteqdot} \phi$, for any valuations α and β. Therefore, we write $A \models^T_{\sim,\doteqdot} \phi$ without α.

The main results of this chapter regarding the relationship between testing satisfaction and behavioural satisfaction are presented in the following theorems. Let A be a Σ-algebra and $\alpha : X \to Dom(\approx_A)$ be a valuation. Let $\sim$ and $\doteqdot$ be Σ-approximate equalities.

Theorem 3.9 *Let $\sim$ be a complete equality, $\doteqdot$ be a sound equality and ϕ be a Σ-formula. If ϕ contains only positive occurrences of $\forall$ and negative occurrences of $\exists$, then $A, \alpha \models_{\approx} \phi$ implies $A, \alpha \models^T_{\sim,\doteqdot} \phi$.*

Proof. Let ϕ be transformed into an equivalent Σ-formula Ψ which is obtained from ϕ by applying the following laws:

- The Morgan laws: $\neg(P \wedge Q)$ is equivalent to $\neg P \vee \neg Q$ and $\neg(P \vee Q)$ is equivalent to $\neg P \wedge \neg Q$
- The conditional law: $P \Rightarrow Q$ is equivalent to $\neg P \vee Q$
- The quantifier negation laws: $\neg\exists x \cdot P(x)$ is equivalent to $\forall x \cdot \neg P(x)$ and $\neg\forall x \cdot P(x)$ is equivalent to $\exists x \cdot \neg P(x)$
- The double negation law: $\neg\neg P$ is equivalent to P
- The bi-conditional law: $P \Leftrightarrow Q$ is equivalent to $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

such that for all sub-formula $\neg\psi$ of Ψ, ψ is an equation. This removes all occurrences of $\Leftrightarrow$ and $\Rightarrow$ and converts negative $\exists$ to positive $\forall$. Since each $\exists$ is negative, this removes all of them.

The proof is conducted by induction on the structure of Ψ.

$(i)\Psi \equiv t = t'$**.** Suppose $\alpha^{\#}(t) \approx_A \alpha^{\#}(t')$. As $t = t'$ is in positive position and $\sim$ is complete, then $\alpha^{\#}(t) \sim_A \alpha^{\#}(t')$.

$(ii)\Psi \equiv \neg\psi$**.** We need to show that $A, \alpha \models_{\approx} \neg\psi$ implies $A, \alpha \models^{T}_{\sim,\doteqdot} \neg\psi$, that is, $A, \alpha \models_{\approx} \psi$ does not hold implies $A, \alpha \models^{T}_{\doteqdot,\sim} \psi$ does not hold. We prove the contrapositive. Suppose $A, \alpha \models^{T}_{\doteqdot,\sim} \psi$. As ψ is an equation $t = t'$ and $\neg t = t'$ is always in positive position, $\alpha^{\#}(t) \doteqdot_A \alpha^{\#}(t')$. By Lemma 2.2, $\alpha^{\#}(t), \alpha^{\#}(t') \in Dom(\approx_A)$. Since $\doteqdot$ is sound, $\alpha^{\#}(t) \approx_A \alpha^{\#}(t')$. Thus, $A, \alpha \models_{\approx} \psi$.

$(iii)\Psi \equiv \psi_1 \wedge \psi_2$**.** Suppose $A, \alpha \models_{\approx} \psi_1 \wedge \psi_2$. Then, by definition, $A, \alpha \models_{\approx} \psi_1$ and $A, \alpha \models_{\approx} \psi_2$. By induction hypothesis, $A, \alpha \models^{T}_{\sim,\doteqdot} \psi_1$ and $A, \alpha \models^{T}_{\sim,\doteqdot} \psi_2$. Thus, $A, \alpha \models^{T}_{\sim,\doteqdot} \psi_1 \wedge \psi_2$.

$(iii)\Psi \equiv \psi_1 \vee \psi_2$ Suppose $A, \alpha \models_{\approx} \psi_1 \vee \psi_2$. Then, by definition, either $A, \alpha \models_{\approx} \psi_1$ or $A, \alpha \models_{\approx} \psi_2$. Suppose $A, \alpha \models_{\approx} \psi_1$. By induction hypothesis, $A, \alpha \models^{T}_{\sim,\doteqdot} \psi_1$. Thus, $A, \alpha \models^{T}_{\sim,\doteqdot} \psi_1 \vee \psi_2$. Suppose $A, \alpha \models_{\approx} \psi_2$. By induction hypothesis, $A, \alpha \models^{T}_{\sim,\doteqdot} \psi_2$. Hence, $A, \alpha \models^{T}_{\sim,\doteqdot} \psi_1 \vee \psi_2$.

$(vi)\Psi \equiv \forall x : s \cdot \psi$**.** Suppose $A, \alpha \models_{\approx} \forall x : s \cdot \psi$. Then $A, \alpha[x \mapsto v] \models_{\approx} \psi$ for any $v \in Dom(\approx_A)$. By induction hypothesis, $A, \alpha[x \mapsto v] \models^T_{\sim,\doteq} \psi$. So, $A, \alpha[x \mapsto v] \models^T_{\sim,\doteq} \psi$ holds for all $v \in Dom(\approx_A)$. But, $^{\#}(T) \subseteq Dom(\approx_A)$, by Definition 2.5 and Lemma 2.2. Hence, $A, \alpha \models^T_{\sim,\doteq} \forall x : s \cdot \psi$.

□

Theorem 3.9 is a generalisation of the results established by [Bernot, 1989] for positive conditional specifications with conditions of observable sorts.[1] There, the only negative occurrences of equations are in the conditions and the restriction to observable equations is so that the computational equality, which is sound and complete, can be used. The black-box equality, which is complete but need not be sound, is used in the conclusion, which is a positive occurrence. Finally, the only quantifiers in conditional equations are universal quantifiers outermost, that is, in positive position. As in the context investigated by Bernot, Theorem 3.9, which covers a prevalent use of $\forall$ and $\exists$, implies that incorrect programs can be accepted by an approximate oracle, but failure in testing means incorrectness. On the other hand, Theorem 3.10 below, the dual of Theorem 3.9, implies that correct programs can be rejected, but success in testing means correctness. Notice that the assumption on quantifiers in Theorem 3.9 is equivalent to restricting its application to universal formulas only.

Theorem 3.10 *Let $\sim$ be a sound equality, $\doteq$ be a complete equality and ϕ be a Σ-formula. If ϕ contains only negative occurrences of $\forall$ and positive occurrences of $\exists$, then $A, \alpha \models^T_{\sim,\doteq} \phi$ implies $A, \alpha \models_{\approx} \phi$.*

Proof. The proof follows the same pattern as the proof of Theorem 3.9 where a formula Ψ is obtained from ϕ by applying the rules listed there. Regarding quantifiers, Ψ has only positive $\exists$.

$(i)\Psi \equiv t = t'$**.** Suppose $\alpha^{\#}(t) \sim_A \alpha^{\#}(t')$. By Lemma 2.2 and because $\sim$ is sound, $\alpha^{\#}(t) \approx_A \alpha^{\#}(t')$.

$(ii)\Psi \equiv \neg\psi$**.** We need to show that $A, \alpha \models^T_{\sim,\doteq} \neg\psi$ implies $A, \alpha \models_{\approx} \neg\psi$, that is, $A, \alpha \models^T_{\doteq,\sim} \psi$ does not hold implies $A, \alpha \models_{\approx} \psi$ does not hold. We prove the contrapositive. Suppose $A, \alpha \models_{\approx} \psi$. As ψ is an equation and $\doteq$ is complete, then $A, \alpha \models^T_{\doteq,\sim} \psi$.

[1] These results are also presented in [Bernot et al., 1991, Bernot, 1991, Gaudel, 1995].

$(iii)\Psi \equiv \psi_1 \wedge \psi_2$ **and** $(iv)\Psi \equiv \psi_1 \vee \psi_2$ Trivial as in the proof of Theorem 3.9.

$(v)\Psi \equiv \exists x : s \cdot \psi$. Suppose $A, \alpha \models^T_{\sim,\doteq} \exists x : s \cdot \psi$. Suppose $A, \alpha[x \mapsto v] \models^T_{\sim,\doteq} \psi$. By induction hypothesis, $A, \alpha[x \mapsto v] \models_{\approx} \psi$. So, there exists $v \in {}^{\#}(T)_s$ such that $A, \alpha[x \mapsto v] \models_{\approx} \psi$ holds. But ${}^{\#}(T) \subseteq Dom(\approx_A)$ by Definition 2.5 and by Lemma 2.2. Hence, $A, \alpha \models_{\approx} \exists x : s \cdot \psi$.

□

***Example* 3.11** Consider the following INTLIST specification in CASL with the usual sorts, operations and axioms.

spec INTLIST =
  INT
and
  NAT
then
  sort *List*;
  ops *nil* : *List*;
    cons : $Int \times List \to List$;
    head : $List \to Int$;
    tail : $List \to List$;
    take : $Nat \times List \to List$;
    len : $List \to Nat$;
    concat : $List \times List \to List$;

    ...

  axiom $\forall\, n : Nat; l, k : List\ \bullet$
    $(\exists\, s : List\ \bullet\ concat(k, s) = l \wedge len(k) = n) \Rightarrow take(n, l) = k$

    ...

end

where *take* returns a list containing the first n items of a list l, *concat* returns the concatenation of two lists and *len* returns the length of a list. Apart from the one presented above, axioms are omitted for the sake of simplicity.

Note that it is not possible to define an oracle for checking the axiom presented above according to the results obtained by [Bernot, 1989]. There, oracles can only be defined from positive conditional specifications with all conditions (negative positions) of observable sorts. The only quantifiers are universal quantifiers outermost. In the axiom shown above, both quantifiers are present and $concat(k,s) = l$ is a non-observable equation in a negative position. However, by Theorem 3.9, an approximate oracle can be defined, provided that a complete and a sound equality are given. The result presented in Theorem 3.9 leads to similar conclusions as the one presented by Bernot, but it can be applied in a wider context. □

From Theorems 3.12 and 3.13 below, if only exhaustive test sets are considered, no restrictions on quantifiers are necessary. As mentioned in Section 2.3, we are assuming $Dom(\approx_A) = {}^{\#}(T_\Sigma)$.

Theorem 3.12 *Let $\sim$ be a complete equality, $\doteq$ be a sound equality and ϕ be a Σ-formula. Then $A, \alpha \models_{\approx} \phi$ implies $A, \alpha \models^{T_\Sigma}_{\sim,\doteq} \phi$.*

Proof. Follows the same pattern as the proof of Theorem 3.9. □

Theorem 3.13 *Let $\sim$ be a sound equality, $\doteq$ be a complete equality and ϕ be a Σ-formula. Then $A, \alpha \models^{T_\Sigma}_{\sim,\doteq} \phi$ implies $A, \alpha \models_{\approx} \phi$.*

Proof. Follows the same pattern as the proof of Theorem 3.10. □

Even if sound and complete equalities are given, the quantifier problem still needs to be handled. The following are obvious corollaries of Theorems 3.9 and 3.10 respectively.

Corollary 3.14 *Let $\sim$ be a sound and complete equality and ϕ be a Σ-formula. If ϕ contains only positive occurrences of $\forall$ and negative occurrences of $\exists$, then $A, \alpha \models_{\approx} \phi$ implies $A, \alpha \models^{T}_{\sim,\sim} \phi$.*

Proof. From Theorem 3.9. □

Corollary 3.15 *Let $\sim$ be a sound and a complete equality and ϕ be a Σ-formula. If ϕ contains only negative occurrences of $\forall$ and positive occurrences of $\exists$, then $A, \alpha \models^{T}_{\sim,\sim} \phi$ implies $A, \alpha \models_{\approx} \phi$.*

Proof. From Theorem 3.10. □

3.4 Revisiting Test Sets

Even though, from a theoretical point of view, valid and/or unbiased test sets can sort out the quantifier problem (see Theorem 2.8), they impede tests to be performed in general if they are infinite. Nevertheless, if they are finite or can be refined towards finite ones which are valid and/or unbiased, then they can lead to a solution to the oracle problem without requiring the assumptions on quantifiers in Theorems 3.9 and 3.10. In this section, we look into some properties of valid and unbiased test sets.

Definition 2.7 introduces valid and unbiased test sets according to the behavioural testing satisfaction relation given in Definition 2.6. This can be easily modified to consider the testing satisfaction relation given in Definition 3.8 as follows. Exhaustive test sets are represented by the algebra of ground terms as before.

Definition 3.16 (Exhaustive, Valid and Unbiased Σ-test sets) *Let T be a Σ-test set, $\sim, \doteqdot$ be two Σ-approximate equalities and ϕ be a Σ-formula. Let A be a Σ-algebra and $\alpha : X \to Dom(\approx_A)$ be a valuation.*

- T_Σ *is an* exhaustive Σ-*test set;*
- T *is a* valid Σ-test set *for $A, \alpha, \sim, \doteqdot, \phi$ if and only if $A, \alpha \models^{T}_{\sim,\doteqdot} \phi$ implies $A, \alpha \models^{T_\Sigma}_{\sim,\doteqdot} \phi$;*
- T *is an* unbiased Σ-test set *for $A, \alpha, \sim, \doteqdot, \phi$ if and only if $A, \alpha \models^{T_\Sigma}_{\sim,\doteqdot} \phi$ implies $A, \alpha \models^{T}_{\sim,\doteqdot} \phi$;*

Again, for the sake of simplicity, when ϕ is a Σ-sentence, we omit α, saying that T is valid (unbiased) for $A, \sim, \doteqdot, \phi$.

Any Σ-test set T is either valid or unbiased or both, if ϕ has a certain form. This is expressed in Theorems 3.17 and 3.18 as follows. In the sequel, let $\sim, \doteqdot$ be two Σ-approximate equalities. Let A be Σ-algebra and $\alpha : X \to Dom(\approx_A)$.

Theorem 3.17 (Unbiased) *Let ϕ be a Σ-formula. If ϕ has only positive $\forall$ and negative $\exists$, then any Σ-test set T is* unbiased *for $A, \alpha, \sim, \doteqdot, \phi$.*

Proof. We need to show that $A, \alpha \models^{T_\Sigma}_{\sim,\doteqdot} \phi$ implies $A, \alpha \models^{T}_{\sim,\doteqdot} \phi$. Let ϕ be transformed into an equivalent Σ-formula Ψ, as in the proof of Theorem 3.9. Regarding quantifiers, Ψ has only positive $\forall$. The proof is conducted by induction on the structure of Ψ. The only interesting case to look at is $\Psi \equiv \forall x : s \cdot \psi$. Suppose $A, \alpha \models^{T_\Sigma}_{\sim,\doteqdot} \forall x : s \cdot \psi$. Then

$A, \alpha[x \mapsto v] \models^{T_\Sigma}_{\sim,\doteq} \psi$, for any $v \in {}^{\#}(T_\Sigma)_s$. By induction hypothesis, $A, \alpha[x \mapsto v] \models^{T}_{\sim,\doteq} \psi$. Thus, because $T \subseteq T_\Sigma$, by Definition 2.5, $A, \alpha \models^{T}_{\sim,\doteq} \forall x : s \cdot \psi$. □

Theorem 3.18 (Valid) *Let ϕ be a Σ-formula. If ϕ has only negative $\forall$ and positive $\exists$, then any Σ-test set T is valid for $A, \sim, \doteq, \phi$.*

Proof. We need to show that $A, \alpha \models^{T}_{\sim,\doteq} \phi$ implies $A, \alpha \models^{T_\Sigma}_{\sim,\doteq} \phi$. The proof follows the same pattern as the proof of Theorem 3.10 where a formula Ψ is obtained from ϕ. Regarding quantifiers, Ψ has only positive $\exists$. The only interesting case is $\Psi \equiv \exists x : s \cdot \psi$. Suppose $A, \alpha \models^{T}_{\sim,\doteq} \exists x : s \cdot \psi$. Suppose $A, \alpha[x \mapsto v] \models^{T}_{\sim,\doteq} \psi$. By induction hypothesis, $A, \alpha[x \mapsto v] \models^{T_\Sigma}_{\sim,\doteq} \psi$. Thus, because $T \subseteq T_\Sigma$ by Definition 2.5, $A, \alpha \models^{T_\Sigma}_{\sim,\doteq} \exists x : s \cdot \psi$. □

[Le Gall and Arnould, 1996] defines a preorder of test sets so that $T' \leq T$ if and only if T is *more efficient* than T', which means that T accepts a smaller number of programs than T'. There, because they focus on positive conditional specifications, all test sets are unbiased which means that incorrect programs may be accepted but correct programs cannot be rejected. Therefore, the best test sets are the ones which accept a smaller number of incorrect programs. When considering first-order specifications, T and T' may not be unbiased, and then the fact that T leads to accepting a smaller/bigger number of programs than T' does not mean T is "better" than T'. Unbiased test sets are likely to lead to the acceptance of incorrect programs. On the other hand, valid test sets are likely to lead to the rejection of correct programs. Definition 3.19 below presents a relation $T' \leq_{A,\alpha,\sim,\doteq,\phi} T$ (read: T refines T' for $A, \alpha, \sim, \doteq, \phi$) according to whether a testing experiment with T is successful for $A, \sim, \doteq, \phi$ implies that a testing experiment with T' is also successful for $A, \sim, \doteq, \phi$, that is, whether A is accepted when considering T implies that A is accepted when considering T'.

Definition 3.19 (Σ-test sets preorder) *Let T and T' be Σ-test sets and ϕ be a Σ-formula. Then $T' \leq_{A,\alpha,\sim,\doteq,\phi} T$ if and only if $A, \alpha \models^{T}_{\sim,\doteq} \phi$ implies $A, \alpha \models^{T'}_{\sim,\doteq} \phi$. $T' \equiv_{A,\alpha,\sim,\doteq,\phi} T$ if and only if $T' \leq_{A,\alpha,\sim,\doteq,\phi} T$ and $T \leq_{A,\alpha,\sim,\doteq,\phi} T'$.*

Theorem 3.20 below shows that if assumptions on quantifiers are made, then the preorder on test sets coincides with subset or superset relations. It is easy to check that this theorem do not hold for the general case where test sets are unbiased (valid) and assumptions on quantifiers of Theorem 3.17 (3.18) are not met.

Theorem 3.20 *Let T and T' be Σ-test sets such that $T' \subseteq T$.*

1. *If ϕ has only positive $\forall$ and negative $\exists$, then $T' \leq_{A,\alpha,\sim,\simeq,\phi} T$;*
2. *If ϕ has only negative $\forall$ and positive $\exists$, then $T \leq_{A,\alpha,\sim,\simeq,\phi} T'$.*

Proof. The proof is conducted by induction on the structure of ϕ. The only interesting cases to look at are when quantifiers are involved. Let ϕ be transformed into an equivalent Σ-formula Ψ as in the proof of Theorem 3.9.

1. We need to show that $A, \alpha \models^{T}_{\sim,\simeq} \Psi$ implies $A, \alpha \models^{T'}_{\sim,\simeq} \Psi$.
   - $\Psi \equiv \forall x : s \cdot \psi$. Suppose $A, \alpha \models^{T}_{\sim,\simeq} \forall x : s \cdot \psi$. Then $A, \alpha[x \mapsto v] \models^{T}_{\sim,\simeq} \psi$, for any $v \in {}^{\#}(T)_s$. By induction hypothesis, $A, \alpha[x \mapsto v] \models^{T'}_{\sim,\simeq} \psi$. Thus, because $T' \subseteq T$, $A, \alpha \models^{T'}_{\sim,\simeq} \forall x : s \cdot \psi$.
2. We need to show that $A, \alpha \models^{T'}_{\sim,\simeq} \Psi$ implies $A, \alpha \models^{T}_{\sim,\simeq} \Psi$.
   - $\Psi \equiv \exists x : s \cdot \psi$. Suppose $A, \alpha \models^{T'}_{\sim,\simeq} \exists x : s \cdot \psi$. Suppose $A, \alpha[x \mapsto v] \models^{T'}_{\sim,\simeq} \psi$. By induction hypothesis, $A, \alpha[x \mapsto v] \models^{T}_{\sim,\simeq} \psi$. Thus, because $T' \subseteq T$, $A, \alpha \models^{T}_{\sim,\simeq} \exists x : s \cdot \psi$.

□

Under the conditions stated in Theorem 3.20, we can conclude that the bigger an unbiased test set is the more accurate it is. In other words, this means that a smaller number of incorrect programs are accepted than when considering its subsets. Likewise, the bigger a valid test set is the more accurate it is, that is, a smaller number of correct programs are rejected than when considering its subsets.

Test sets that are both valid and unbiased are "equivalent" in the sense that they can always substitute the exhaustive test set and neither incorrect programs are accepted nor correct programs are rejected.

In general, oracles can be defined without assumptions on test sets, but the converse is not always true. For instance, test sets are defined as valid and unbiased w.r.t. a given satisfaction relation (see Definitions 2.7 and 3.16). From Theorem 3.17, T is always unbiased in the context of Theorem 3.9 due to the absence of positive $\exists$. Therefore, an assumption requiring T to be unbiased can replace the restrictions on quantifiers. This would make it possible to define an oracle when a given axiom has both positive $\forall$ and positive $\exists$ and obtain the same conclusions as the ones in Theorem 3.9. As mentioned before, we are assuming $Dom(\approx_A) =^{\#} (T_\Sigma)$.

Theorem 3.21 *Let $\sim$ be a complete equality, $\simeq$ be a sound equality and ϕ be a Σ-formula. If T is unbiased for $A, \alpha, \sim, \simeq, \phi$, then $A, \alpha \models_{\approx} \phi$ implies $A, \alpha \models^{T}_{\sim,\simeq} \phi$.*

Proof. Suppose $A, \alpha \models_{\approx} \phi$. By Theorem 3.12, $A, \alpha \models^{T_\Sigma}_{\sim,\simeq} \phi$. Since T is unbiased for $A, \alpha, \sim, \simeq, \phi$, then $A, \alpha \models^{T}_{\sim,\simeq} \phi$. □

Similarly, assumptions on quantifiers in Theorem 3.10 can be replaced by the validity of T as follows.

Theorem 3.22 *Let $\sim$ be a sound equality, $\simeq$ be a complete equality and ϕ be a Σ-formula. If T is valid for $A, \alpha, \sim, \simeq, \phi$, then $A, \alpha \models^{T}_{\sim,\simeq} \phi$ implies $A, \alpha \models_{\approx} \phi$.*

Proof. Suppose $A, \alpha \models^{T}_{\sim,\simeq} \phi$. Since T is valid for $A, \alpha, \sim, \simeq, \phi$, then $A, \alpha \models^{T_\Sigma}_{\sim,\simeq} \phi$. Hence, by Theorem 3.13, $A, \alpha \models_{\approx} \phi$. □

Theorems 3.21 and 3.22 can be seen as special cases of the general correctness Theorem 2.8 given in Chapter 2, where two approximate equalities are used instead of a single behavioural equality.

3.5 Revisiting Approximate Equalities

As valid and unbiased test sets are defined in terms of a Σ-algebra, a valuation, a pair of approximate equalities and a Σ-formula in Definition 3.16, it is reasonable to think that given A, α, ϕ, a number of possible pairs of equalities can make a Σ-test set T be either valid or unbiased. For instance, suppose $\phi = \exists x \cdot \phi'$. Depending on the equality procedures applied inside ϕ' a witness x might be found or not among the values of T.

Definition 3.23 (Valid and Unbiased Pair of Equalities) *Let T be a Σ-test set, $\sim, \simeq$ be two Σ-approximate equalities and ϕ be a Σ-formula. Let A be a Σ-algebra and $\alpha : X \to Dom(\approx_A)$.*

- *$\sim, \simeq$ is a* valid *pair of equalities for A, α, ϕ, T if and only if $A, \alpha \models^{T}_{\sim,\simeq} \phi$ implies $A, \alpha \models^{T_\Sigma}_{\sim,\simeq} \phi$;*
- *$\sim, \simeq$ is an* unbiased *pair of equalities for A, α, ϕ, T if and only if $A, \alpha \models^{T_\Sigma}_{\sim,\simeq} \phi$ implies $A, \alpha \models^{T}_{\sim,\simeq} \phi$;*

Obviously, if $\sim, \doteq$ is a valid pair of equalities for A, α, ϕ, T, then T is valid for $A, \alpha, \sim, \doteq, \phi$. Also, if $\sim, \doteq$ is a unbiased pair of equalities for A, α, ϕ, T, then T is unbiased for $A, \alpha, \sim, \doteq, \phi$.

Definition 3.23 brings out the fact that the quantifier problem may demand equalities which also make test sets be either unbiased or valid. For instance, the effort to define finite procedures for implementing behavioural equalities may not pay off if T is neither unbiased nor valid. Approximate equalities give more flexibility in dealing with both the equality and quantifier problems since they are adjustable within the constraints of soundness/completeness with respect to behavioural equality instead of being fixed. Thus they can be selected to give valid/unbiased test sets as well as giving a reasonably accurate approximation to behavioural equality.

Theorems 3.24 and 3.25 below are the counterparts of Theorems 3.21 and 3.22, when valid/unbiased pair of equalities are considered.

Theorem 3.24 *Let $\sim$ be a complete equality, $\doteq$ be a sound equality and ϕ be a Σ-formula. If $\sim, \doteq$ is unbiased for A, α, ϕ, T, then $A, \alpha \models_{\approx} \phi$ implies $A, \alpha \models^{T}_{\sim,\doteq} \phi$.*

Proof. From Theorem 3.21 since T is unbiased for $A, \alpha, \sim, \doteq, \phi$. □

Theorem 3.25 *Let $\sim$ be a sound equality, $\doteq$ be a complete equality and ϕ be a Σ-formula. If $\sim, \doteq$ is valid for A, α, ϕ, T, then $A, \alpha \models^{T}_{\sim,\doteq} \phi$ implies $A, \alpha \models_{\approx} \phi$.*

Proof. From Theorem 3.22 since T is valid for $A, \alpha, \sim, \doteq, \phi$. □

Even if it is not possible to give an implementation of behavioural equality, it may be interesting to refine approximate equalities such that more accurate results can be achieved. However, this may affect test sets making them become either invalid or biased. A preorder on pairs of equality $(\sim', \doteq') \leq_{A,\alpha,\phi,T} (\sim, \doteq)$ (read: $(\sim, \doteq)$ refines $(\sim', \doteq')$ for A, α, ϕ, T) is defined as follows.

Definition 3.26 (Pair of Equalities preorder) *Let $\sim', \doteq'$ and $\sim, \doteq$ be two pairs of Σ-approximate equalities, T be a Σ-test set and ϕ be a Σ-formula. Let A be a Σ-algebra and $\alpha : X \to Dom(\approx_A)$. Then $(\sim', \doteq') \leq_{A,\alpha,\phi,T} (\sim, \doteq)$ if and only if $A, \alpha \models^{T}_{\sim,\doteq} \phi$ implies $A, \alpha \models^{T}_{\sim',\doteq'} \phi$. Also, $(\sim', \doteq') \equiv_{A,\alpha,\phi,T} (\sim, \doteq)$ if and only if $(\sim', \doteq') \leq_{A,\alpha,\phi,T} (\sim, \doteq)$ and $(\sim, \doteq) \leq_{A,\alpha,\phi,T} (\sim', \doteq')$.*

Theorem 3.27 *Let $\sim'$, $\bumpeq'$, $\sim$ and $\bumpeq$ be Σ-approximate equalities such that $\sim'_A \subseteq \sim_A$ and $\bumpeq'_A \supseteq \bumpeq_A$. Then $(\sim, \bumpeq) \leq_{A,\alpha,\phi,T} (\sim', \bumpeq')$.*

Proof. We need to show that $A, \alpha \models^{T}_{\sim',\bumpeq'} \phi$ implies $A, \alpha \models^{T}_{\sim,\bumpeq} \phi$. The proof follows the same pattern as the proof of Theorem 3.9 where a formula Ψ is obtained from ϕ by applying the rules listed there. The only interesting cases to look at are when Ψ is an equation or the negation of an equation.

$(i)\Psi \equiv t = t'$**.** Suppose $\alpha^{\#}(t) \sim'_A \alpha^{\#}(t')$. Since $\sim'_A \subseteq \sim_A$, $\alpha^{\#}(t) \sim_A \alpha^{\#}(t')$.

$(ii)\Psi \equiv \neg\psi$**.** As ψ is an equation, we need to show that $\alpha^{\#}(t) \bumpeq_A \alpha^{\#}(t')$ implies $\alpha^{\#}(t) \bumpeq'_A \alpha^{\#}(t')$. Suppose $\alpha^{\#}(t) \bumpeq_A \alpha^{\#}(t')$. Since $\bumpeq'_A \supseteq \bumpeq_A$, then $\alpha^{\#}(t) \bumpeq'_A \alpha^{\#}(t')$.

$\square$

Theorem 3.27 says that if $\sim'$ is finer than $\sim$ and $\bumpeq'$ is coarser than $\bumpeq$, then a smaller number of programs may be accepted when the pair $\sim', \bumpeq'$ is considered instead of $\sim, \bumpeq$. This means that if $\sim', \sim$ are complete and $\bumpeq', \bumpeq$ are sound, an oracle would accept fewer incorrect programs when the pair $\sim', \bumpeq'$ is considered instead of $\sim, \bumpeq$, whereas if $\sim', \sim$ are sound and $\bumpeq', \bumpeq$ are complete, an oracle would reject fewer correct programs when the pair $\sim, \bumpeq$ is considered instead of $\sim', \bumpeq'$. Therefore, as one would have expected, finer complete equalities and coarser sound equalities must be defined whenever possible. Nevertheless, the new equalities may not preserve the fact that T is either unbiased or valid. For instance, if $\sim'_A \subseteq \sim_A$ and $\bumpeq'_A \supseteq \bumpeq_A$ so that $(\sim, \bumpeq) \leq_{A,\alpha,\phi,T} (\sim', \bumpeq')$ and $\sim', \bumpeq'$ is unbiased for A, α, ϕ, T, it does not imply that $\sim, \bumpeq$ is also unbiased for A, α, ϕ, T as illustrated below.

$$\begin{tikzcd}
A, \alpha \models^{T_\Sigma}_{\sim',\bumpeq'} \phi \arrow[rr, Rightarrow, "{\sim', \bumpeq' \text{ is unbiased}}"] \arrow[d, Rightarrow, "{\sim' \subseteq \sim \text{ and } \bumpeq' \supseteq \bumpeq}"'] & & A, \alpha \models^{T}_{\sim',\bumpeq'} \phi \arrow[d, Rightarrow, "{\sim' \subseteq \sim \text{ and } \bumpeq' \supseteq \bumpeq}"] \\
A, \alpha \models^{T_\Sigma}_{\sim,\bumpeq} \phi \arrow[rr, Rightarrow, "{\text{is } \sim, \bumpeq \text{ is unbiased ?}}"] & & A, \alpha \models^{T}_{\sim,\bumpeq} \phi
\end{tikzcd}$$

The answer is likely to be no. For instance, consider $\phi \equiv \exists x : s \cdot \psi$. There might be a witness $v \in T_\Sigma$ whereby, although $A, \alpha \models^{T_\Sigma}_{\sim',\bumpeq'} \phi$ is not satisfied, $A, \alpha \models^{T_\Sigma}_{\sim,\bumpeq} \phi$

holds, but v might not be in T. Thus, $A,\alpha \models^{T}_{\sim,\bumpeq} \phi$ does not hold. Likewise, consider $\sim'_A \supseteq \sim_A$ and $\bumpeq'_A \subseteq \bumpeq_A$ so that $(\sim',\bumpeq') \leq_{A,\alpha,\phi,T} (\sim,\bumpeq)$.

$$
\begin{tikzcd}
A,\alpha \models^{T_\Sigma}_{\sim',\bumpeq'} \phi \arrow[r, Rightarrow, "{\sim',\bumpeq' \text{ is unbiased}}"] & A,\alpha \models^{T}_{\sim',\bumpeq'} \phi \\
A,\alpha \models^{T_\Sigma}_{\sim,\bumpeq} \phi \arrow[u, Rightarrow, "{\sim'\supseteq\sim \text{ and } \bumpeq'\subseteq\bumpeq}"] \arrow[r, Rightarrow, "{\text{is } \sim,\bumpeq \text{ is unbiased ?}}"] & A,\alpha \models^{T}_{\sim,\bumpeq} \phi \arrow[u, Rightarrow, "{\sim'\supseteq\sim \text{ and } \bumpeq'\subseteq\bumpeq}"']
\end{tikzcd}
$$

Again the answer is likely to be no. For instance, consider $\phi \equiv \exists x : s \cdot \psi$. There might be a witness $v' \in T$ such that $A,\alpha \models^{T}_{\sim',\bumpeq'} \phi$, but $A,\alpha \models^{T}_{\sim,\bumpeq} \phi$ does not hold.

In summary, it may not be appropriate to refine approximate equalities towards behavioural equalities, since test sets may thereby become biased or invalid. However, from Theorem 3.17, if ϕ has only positive $\forall$ and negative $\exists$, T is unbiased for any $A,\sim,\bumpeq$. Also, from Theorem 3.18, if ϕ has only negative $\forall$ and positive $\exists$, T is valid for any $A,\sim,\bumpeq$. In more general cases where quantifiers in both positive and negative positions are present, one may need to ponder on the benefits of either having valid and/or unbiased test sets or closer approximations to behavioural equality, unless it is possible to achieve both of them.

3.6 Grey-Box Approach

Results obtained from Sections 3.3 to 3.5 (see Figure 3.2) show how the equality problem can be sorted out by using two approximate equalities and also how the quantifier problem can be handled. In this section, a way of applying these results is proposed, namely the grey-box approach.

The grey-box approach combines the white-box and black-box approach in order to define an approximate oracle with tests performed according to Definition 3.8, where the white-box approach is used to produce a sound equality $\simeq$ and the black-box approach is used to produce a complete equality $\eqsim$ from a finite subset of observable contexts. The reason for defining two equalities is that each one can be successfully applied in contexts where the other might not be, according to Theorems 3.9 and 3.10 and their variants. A combination of white-box and black-box techniques to

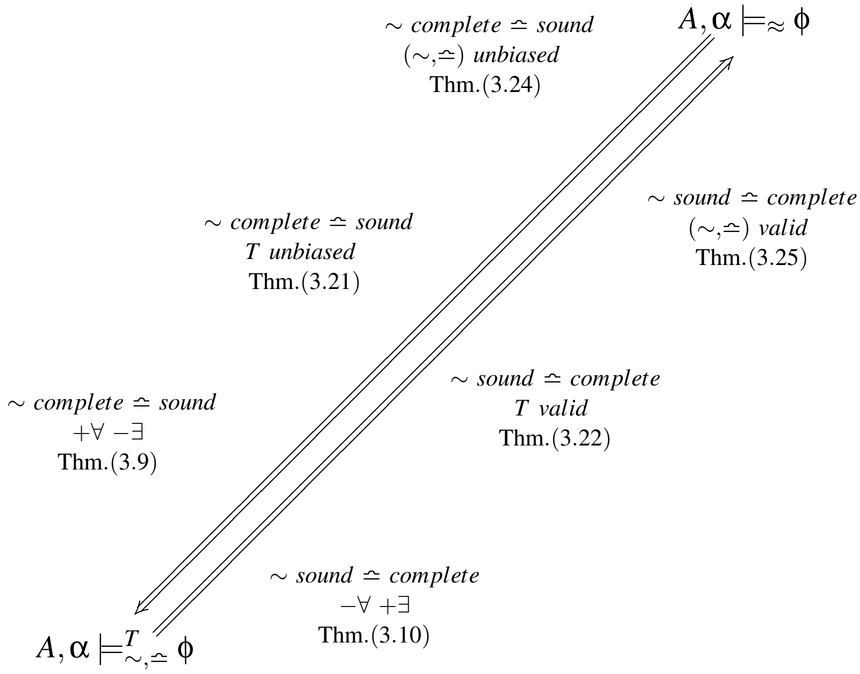

Figure 3.2: Approximate oracle and correctness.

sort out the equality problem can also be found in [Doong and Frankl, 1994, Chen et al., 1998].

Equalities defined from a subset of the set of all observable contexts – contextual equalities – are always complete w.r.t. observational equality. Thus, the black-box approach can be used to find an axiomatisation over the crucial contexts[2] (see Section 2.4). The equality induced by these contexts either coincides with the observational equality or is a complete approximate equality.

Proposition 3.28 *Let $C \subseteq C_{Obs}$. The contextual equality $\sim_{C,A}$ is complete.*

Proof. $Dom(\approx_{Obs,A}) = Dom(\sim_{C,A})$. Obviously, $\approx_{Obs,A} \subseteq \sim_{C,A}$. $\square$

Sound and complete equalities approximate the behavioural equality from opposite directions ($\simeq \subseteq \approx \subseteq \eqsim$). Whenever the black-box equality is complete but not sound, it is reasonable to define a sound equality by using the white-box approach. Structural equalities based on the equality of the values of the concrete representation of a sort s, even though not always complete, are always sound. However, as structural equalities are essentially white-box, it is more convenient to formalise them in the

[2] $f(x_1 \ldots z_{s_k} \ldots x_n)$, with $f : s_1 \ldots s_k \ldots s_n \to s \in F$, $s \in Obs$ and $s_k \in S \setminus Obs$

more concrete level of programming languages where datatypes can be defined rather than in the level of algebras which are basically composed of values and functions. Sound equalities can be defined in a number of ways, possibly relying on intuition. The most simple and straightforward sound equality is the *set-theoretical* equality which is a reflexive relation where a equals a' if and only if a, a' denote the same value.

Proposition 3.29 *The set-theoretical equality is sound.*

The closer to a behavioural equality an approximate equality is, the more effective a testing experiment is regarding the detection of correctness (see Section 3.5). However, it is also vital to consider test sets when refining approximate equalities as discussed in Section 3.5. In practice, an obvious way of moving towards soundness, is to add new predicates to a equality axiomatisation using the $\wedge$ connective. This can make the equality relate a smaller number of values than before. Conversely, in order to move towards completeness, new predicates can be appended using the $\vee$ connective. This can make the equality relates a greater number of values than before. But, obviously, there is always a risk of losing completeness/soundness. Furthermore, given that valid and/or unbiased test sets are infinite, there is no recipe to find subsets which are also valid and/or unbiased. However, finite supersets can be more effective (see Theorem 3.20).

The grey-box approach can be applied with the following purposes.

(i) Attempt to detect incorrectness without rejecting correct programs. This is based on Theorems 3.9, 3.12, 3.21 and 3.24 and consists in applying the sound equality in negative occurrences of equations and the complete equality in the positive ones.

(ii) Attempt to detect correctness without accepting incorrect programs. This is based on Theorems 3.10, 3.13, 3.22 and 3.25 and consists in applying the complete equality in negative occurrences of equations and the sound equality in the positive ones.

Depending on the alternative chosen, different conclusions about correctness and incorrectness can be achieved. Whenever testing is not successful in (i) we can conclude that the program is incorrect. However, it is easy to check that the converse does not hold: if testing is successful we cannot conclude that the program is correct. On the other hand, whenever testing is successful in (ii) we can conclude that the program is correct, but once again the converse does not hold, that is, if testing is not successful, we cannot conclude the program is incorrect.

Approach (i) is the classical one to testing whereas approach (ii) is an innovation in testing theory and practice: detecting correctness. Whenever the premises of one of Theorems 3.9, 3.12, 3.21 or 3.24 and of its respective dual are satisfied so that equalities can be applied in both positive and negative positions, then both approaches (i) and (ii) can be tried in order to achieve a higher degree of confidence in the interpretation given by the oracle. For instance, if the test fails in approach (ii) and, thereafter, in (i), the program is incorrect while if it succeeds in approach(i) and, thereafter, in (ii), then the program is correct.

Example 3.30 below concludes this chapter by illustrating the grey-box approach applied to a specification of the unification algorithm.

***Example* 3.30 (Unification)** Unification, which plays a central role in theorem-proving, is the process of finding a common instance of two expressions, and if such an instance exists, the algorithm produces a substitution that yields that instance. The unification specification presented here is modelled on the one presented in [Manna and Waldinger, 1981].

Firstly, an specification of expressions, namely EXPRESSION, is given as follows. An expression can be a constant, a variable or a function application to a list of expressions. The FUNCTION, VAR and CONST specifications model functions, variables and constants respectively.

spec FUNCTION =
  NAT
and
  STRING
then
  sort *function*;
  ops $declare \; : \; String \times Nat \to function;$
    $arity \; : \; function \to Nat;$
  vars $n \; : \; String;$
    $i \; : \; Nat$
  • $arity(declare(n,i)) \; = \; i$
end

spec VAR =
  CHAR
with
  sort *Char* $\mapsto$ *variable*
end

spec CONST =
  CHAR
with
  sort *Char* $\mapsto$ *constant*
end

spec EXP =
  sort *expression*;
end

spec EXPRESSION =
  FUNCTION **and**
  CONST **and**
  LIST [EXP **fit sort** Elem $\mapsto$ expression] **and**
  FINITESET [VAR **fit sort** Elem $\mapsto$ variable]
then

  ops mk_c : $constant \to expression$;
    mk_v : $variable \to expression$;
    mk_f : $function \times List[expression] \to expression$;
    $variables$: $expression \to FinSet[variable]$;
    $variablesl$: $List[expression] \to FinSet[variable]$;

  preds is_valid : $expression$;
    is_validl : $List[expression]$;

  vars c : $constant$;
    v : $variable$;
    f : $function$;
    e : $expression$;
    l : $List[expression]$

- $is\_valid(mk\_c(c))$
- $is\_valid(mk\_v(v))$
- $is\_valid(mk\_f(f,l)) \Leftrightarrow \#l = arity(f) \wedge is\_validl(l)$
- $is\_validl(nil)$

- $is\_validl(e :: l) \;\Leftrightarrow\; is\_valid(e) \;\wedge\; is\_validl(l)$
- $variables(mk\_c(c)) \;=\; \{\}$
- $variables(mk\_v(v)) \;=\; set(v)$
- $variables(mk\_f(f,l)) \;=\; variablesl(l)$
- $variablesl(nil) \;=\; \{\}$
- $variablesl(e :: l) \;=\; (variables(e))\ union\ (variablesl(l))$

hide
  variablesl, *is_validl*
end

where LIST[ELEM] and FINITESET[ELEM] are generic specifications of sets and lists respectively with the usual operations. An expression is valid when functions are applied to the correct number of arguments.

Substitution is an operation that replaces specific variables of an expression by other expressions. The SUBSTITUTION specification is presented below as an extension of EXPRESSION.

spec SUBSTITUTION =
  EXPRESSION
then
  sort *substitution*;
  ops
  $empty \;:\; substitution;$
  $dom \;:\; substitution \rightarrow FinSet[variable];$
  $rng \;:\; substitution \rightarrow FinSet[variable];$
  $add\_s \;:\; variable \times expression \times substitution \rightarrow substitution;$
  $apply \;:\; expression \times substitution \rightarrow expression;$
  $applyl \;:\; List[expression] \times substitution \rightarrow List[expression];$
  $compose \;:\; substitution \times substitution \rightarrow substitution;$

  vars
  $x \;:\; variable;$
  $e \;:\; expression;$
  $s \;:\; substitution$

- $dom(empty) \;=\; \{\} \;\wedge\; rng(empty) \;=\; \{\}$
- $\neg\,(x\ elemOf\ dom(s)) \;\wedge\; \neg\,(e \;=\; mk\_v(x)) \;\Rightarrow$
  $dom(add\_s(x,e,s)) \;=\; (set(x))\ union\ (dom(s)) \;\wedge$
  $rng(add\_s(x,e,s)) \;=\; (variables(e))\ union\ (rng(s))$

vars c : *constant*;
x, y : *variable*;
f : *function*;
e : *expression*;
s, s' : *substitution*;
l : *List*[*expression*]

- $apply(e, empty) \ = \ e$
- $apply(mk\_c(c), s) \ = \ mk\_c(c)$
- $apply(mk\_v(x), add\_s(x, e, s)) \ = \ e$
- $\neg\,(x \ = \ y) \ \Rightarrow \ apply(mk\_v(y), add\_s(x, e, s)) \ = \ apply(mk\_v(y), s)$
- $apply(mk\_f(f, l), s) \ = \ mk\_f(f, applyl(l, s))$
- $applyl(nil, s) \ = \ nil$
- $applyl(e :: l, s) \ = \ (apply(e, s)) :: (applyl(l, s))$
- $apply(e, compose(s, s')) \ = \ apply(apply(e, s), s')$

vars s, s', s'' : *substitution*

- $compose(s, empty) \ = \ s \ \wedge \ compose(empty, s) \ = \ s$
- $compose(compose(s, s'), s'') \ = \ compose(s, compose(s', s''))$

hide

applyl

end

where the second axiom says that if x is not in $dom(s)$ and e is an expression distinct from $mk\_v(x)$, then the result of adding the replacement "e for x" to the substitution s is also a substitution $add\_s(x, e, s)$ such that the conditions showed there hold.

The UNIFICATION specification is given as follows as an extension of the SUBSTITUTION specification.

spec UNIFICATION =

SUBSTITUTION

then

op *unify* : $expression \times expression \rightarrow substitution$;

preds *is_moregeneral* : $substitution \times substitution$;
is_idempotent : *substitution*;

axiom $\forall\, e, e' : expression \bullet$
$(\exists\, s : substitution \bullet apply(e,s) = apply(e',s)) \Rightarrow$
$apply(e, unify(e,e')) = apply(e', unify(e,e')) \wedge$
$(\forall\, s' : substitution \bullet$
$apply(e,s') = apply(e',s') \Rightarrow$
$is\_moregeneral(unify(e,e'),s')) \wedge$
$is\_idempotent(unify(e,e'))$
axiom $\forall\, s, s' : substitution \bullet$
$is\_moregeneral(s,s') \Leftrightarrow$
$(\exists\, r : substitution \bullet s' = compose(s,r))$
axiom $\forall\, s : substitution \bullet$
$is\_idempotent(s) \Leftrightarrow dom(s)\ intersection\ rng(s) = \{\}$

hide

is_moregeneral, *is_idempotent*

end

Let *constant*, *variable* and *function* be observable sorts. An oracle for the *unify* axiom (first axiom in UNIFICATION) needs to compute equality on *expression*. Suppose that *expression* is a non-observable sort. The black-box approach is not appropriate to directly define an observational equality on *expression*, because its crucial contexts ($is\_valid(z_{exp})$,$vars(z_{exp})$) are not descriptive enough and then hidden functions must be added. Thus, the grey-box approach seems to be a better choice. It is easy to see that the unify axiom has only positive $\forall$ and negative $\exists$. Then, based on approach (i) (presented in this section) and Theorem 3.9, we define a complete equality on expressions (for positive positions) using the black-box approach and a sound equality (for negative positions) using the white-box approach. This indicates that only incorrectness of the implementation can be detected. An implementation of the lift of EXPRESSION is given as follows, where equality on *expression* is implemented by `eqs` which is clearly a structural sound equality.

```
signature LExpression =
sig
  include Expression;
  val eqs : expression * expression -> bool;
end;

functor LExpression (structure function : Function
                     structure const : Const
                     structure var : Var ) :
  sig
    include LExpression
```

```
    sharing F = function and C = const and V = var
  end =
struct
  ...
  datatype expression =
     mk_c of C.constant |
     mk_v of V.variable |
     mk_f of F.function * (expression list)
  ...
  fun eqs (mk_c(c),mk_c(c')) = c = c'
    | eqs (mk_v(v),mk_v(v'))   = v = v'
    | eqs (mk_f(f,l),mk_f(f',l')) =
               f = f' andalso eqsl(l,l')
    | eqs (e,e') = false
  and eqsl([],[]) = true
    | eqsl(e::l,e'::l') = eqs(e,e') andalso eqsl(l,l')
    | eqsl(l,l') = false
end
```

For sake of clarity, the definitions of `is_valid`, `variables` and `variablesl` are omitted in the presentation above. Obviously, different implementations of `eqs` may be defined from different implementations of `expression`.

A complete equality defined from the finite set of crucial contexts of `expression` is given as follows.

$$\forall e, e' \cdot eqc(e, e') = (is\_valid(e) = is\_valid(e')) \wedge (vars(e) = vars(e'))$$

Finally, an oracle for the unification axiom can be defined as a boolean function, where, according to Theorem 3.9, `true` means *undefined*, as the program may be either correct or incorrect, and `false` means *incorrect*. This function, `unify_o`, is as follows, where `le` and `ls` are lists of expressions and substitutions respectively which constitute the test set.

```
fun unify_o (le,ls) =
 o_forall le (fn e => o_forall le (fn e' =>
  o_exists ls (fn s =>
   eqs(apply(e,s),apply(e',s))) implies
    ( (eqc(apply(e,unify(e,e')),apply(e',unify(e,e'))))
           andalso
      (o_forall ls (fn s' =>
         eqs(apply(e,s'),apply(e',s')) implies
            is_moregeneral(unify(e,e'),s')))
```

```
          andalso
      is_idempotent(unify(e,e')) ) ))
```

Now, consider how an oracle for the other axioms of UNIFICATION can be defined. In this case, it is necessary to compare values of *substitution* in order to check the axioms. Let *substitution* and *expression* be non-observable sorts (see the end of the section for the case where *expression* is observable). The crucial contexts on *substitution* are $dom(z_{subst})$, $rng(z_{subst})$, $is\_moregeneral(z_{subst}, s)$, $is\_moregeneral(s, z_{subst})$ and $is\_idempotent(z_{subst})$. However, it is clear that substitutions are equal if they produce the same result when applied to all expressions. As *expression* is a non-observable sort, *apply* cannot be used as a crucial context. Thus, as the set of crucial contexts is not enough to construct a sound and complete equality, the grey-box approach seems to be a better choice again.

Consider the axiom which specifies the *is_moregeneral* operation. This axiom can be converted to the following two by expanding $\Leftrightarrow$.

$$[\psi]\ \forall s, s' : substitution \centerdot \quad is\_moregeneral(s, s') \Rightarrow (\exists\, r : substitution \centerdot s' = compose(s, r))$$

$$[\rho]\ \forall s, s' : substitution \centerdot \quad (\exists\, r : substitution \centerdot s' = compose(s, r)) \Rightarrow is\_moregeneral(s, s')$$

Here we have a positive occurrence of $\forall$ and a positive and a negative occurrence of $\exists$ and then neither Theorem 3.9 nor 3.10 is applicable for both ψ and ρ. Whenever quantified formulas occur as one side of *iff* predicates, these theorems cannot be applied in both directions of the bi-conditional. The direction which does not meet the premises of the theorem being considered must be discarded. It is easy to check that from the standard interpretation of $\wedge$ and Theorem 3.9, $A \models_{\approx} \psi \wedge \rho \Rightarrow A \models_{\approx} \rho \Rightarrow A \models^{T}_{\approx,\simeq} \rho$, but $A \models_{\approx} \psi \Rightarrow A \models^{T}_{\approx,\simeq} \psi$ may not hold unless T is unbiased, due to the positive $\exists$ (see Theorem 3.21). When considering Theorem 3.10, dual conclusions can be reached.

If (ψ) is added to the SUBSTITUTION oracle, then correct programs can be rejected and incorrect programs can be accepted at the same time. The reason for this is that if *is_moregeneral(s,s')* happens to be true, then the test set must have a witness r in order to avoid rejecting a correct program which in general requires either a finite exhaustive or an unbiased test set. In case *is_moregeneral(s,s') = false* and a witness r can be found in the test set, then incorrect programs can be accepted. Hence, only ρ is added to the *is_moregeneral* oracle, implying that incorrect programs can be accepted w.r.t. a simplified UNIFICATION specification without (ψ). This reflects the limitations of testing which is aimed at detecting the presence of errors, but often not all of them can be detected. From Theorem 3.9, a sound equality can be used to compute $s' = compose(s,r)$.

In summary, it is only possible to check ψ if either an unbiased test set or unbiased pair of equalities is given so that Theorems 3.21 and 3.24 can be applied. It is easy to check that if the complete equality accepts all values in the definition domain as equal, any test set for (ψ) is unbiased. Eventually, by varying the complete equality, one might be capable of coming up with a finite and unbiased test set, even though all of them seem to be infinite.

Finally, suppose *expression* is an observable sort. Then the black-box approach seems to be a good choice as $apply(e, z_{subst})$ can be added to the set of crucial contexts. After employing the black-box method presented in Section 2.4, a sound and complete equality on *substitution* can be defined as follows.

$$\forall s, s' \cdot eq(s,s') = \forall e \cdot (apply(e,s) = apply(e,s'))$$

However, notice that, even though the set of observable contexts is finite, an infinite set of expressions is required to compute this equality. Hence, the grey-box approach seems to be a better choice again, since considering a finite and feasible set of expressions will lead to a complete equality definition. Again, Theorem 3.9 can help in interpreting test results as it does not require this equality to be a congruence relation. □

3.7 Concluding Remarks

The oracle problem in the context of algebraic specifications expressed in first-order logic reduces to the general problem of finding an equality procedure for non-observable sorts – the equality problem – and how to deal with quantifiers. This chapter introduces a solution to this problem and an approach to defining decidable oracles, the grey-box approach. First, two classical approaches for handling the equality problem are reviewed. The white-box approach relies on defining equality on the values of a sort by looking at its internal representation. This approach is based on the fact that the result of comparing two values may depend on the particular implementation. Although it is quite feasible, this approach may demand intuition and additional information not present in the abstract specification. The black-box approach is concerned with defining an observational equality and unlike the white-box approach it has a method which covers most of the usual cases and may require a minimum of intuition. Nevertheless, when the “crucial” contexts are not descriptive enough, the method can be difficult to be applied in practice (see Example 3.2) and even if these contexts are sufficient, infinite sets of data may be required (see Example 3.30).

In the grey-box approach, a sound and a complete approximate equality are constructed which are as close as possible or even equal to the behavioural equality. The syntactic position of quantifiers in axioms are taken into account in order to decide in which contexts these equalities are applied. Approximate equalities need not be congruence relations and approximate oracles can be defined independently of test sets. This approach leads to similar levels of confidence as the one presented in [Gaudel, 1995], but it can be applied in a wider context.

Chapter 4

Testing from Structured Specifications

"The design of tests for software and other engineered products can be as challenging as the initial design of the product itself."
R. Pressman

Contents

This chapter investigates the oracle problem in the context of structured algebraic specifications. As we have seen, for flat specifications, this problem reduces to the equality and quantifier problem. However, when focusing on structured specifications, which are composed of a number of specifications with different signatures, this problem can be harder. The reason is that this structure must be taken into account in order to make sense of axioms and define the necessary tests to be performed. Two styles of testing are introduced and contrasted in order to tackle the oracle problem: *flat testing* and *structured testing* [Machado, 2000b]. Under which circumstances test results from both styles indicate behavioural satisfaction and vice-versa is looked into carefully, also instantiating to the specific case of observational satisfaction. Examples are presented to clarify the main ideas and results.

4.1 The Oracle Problem

When testing from a structured specification, it is necessary to think of its structure, even though this structure is not reflected in its models. This is due to the fact that the semantics of specifications is given in a compositional way [Bidoit et al., 1999], that is, the signature and class of models of a specification are determined according to the result of applying specification-building operations to its constituent specifications. Among obstacles that can be encountered when testing from structured specifications are:

1. Any sort can be introduced and/or referred to by operations and axioms in different specifications in the structure. This suggests that a family of equalities on this sort, with one equality for each signature in the structure, may be needed. In other words, it may be necessary to deal with the equality problem for the same sort under different, but related circumstances.

2. Hidden axioms composed of hidden and visible (exported) symbols may describe important properties of operations. But, hidden symbols are not necessarily implemented in the program under test.

3. It is reasonable to think that particular test sets for a given sort should be defined separately for some signatures or groups of axioms. In other words, it may be beneficial to handle the quantifier problem differently according to the signature under consideration at the point where the quantifier appears. Even if a test set is finite, it may be impractical to test certain functions based on this test set, particularly the more complex and time-consuming ones.

Therefore, the oracle problem for structured specifications reduces to the problem of how to deal with the equality and quantifier problems when different signatures and specification-building operations in the structure are involved and also how hidden definitions can be appropriately tackled. Points 1 to 3 listed above are developed in Example 4.1 below and throughout this chapter.

Example* 4.1 (Unification** *continued)** Look once more at the unification specification in Example 3.30. FUNCTION, EXPRESSION, SUBSTITUTION and UNIFICATION are structured specifications constructed by specification-building operations like ***and***, ***then*** and ***hide***, which combines two specifications, extends a specification and hide symbols in a specification respectively [Language Design Group, 2000].

Note that the *substitution* sort is introduced in SUBSTITUTION and incremented in UNIFICATION in the sense that new predicates, which are also observers over this sort, are added. Suppose *expression* is non-observable and let *eqc* be a complete equality on *substitution* defined in the scope of the SUBSTITUTION specification from the set of all crucial contexts of observable sorts.

$$\forall s, s' \cdot eqc(s, s') = (dom(s) = dom(s')) \land (rng(s) = rng(s'))$$

Let UNIFICATION-H be the UNIFICATION specification without the outermost ***hide***. Since the *is_moregeneral* and *is_idempotent* predicates are *substitution* observers, they can be used to define a complete equality *EQC*, in the scope of the UNIFICATION-H specification, differing from *eqc* by the following additional predicates:

$$\begin{array}{l}(is\_idempotent(s) \Leftrightarrow is\_idempotent(s')) \land \\ (\forall r \cdot is\_moregeneral(s, r) \Leftrightarrow is\_moregeneral(s', r)) \land \\ (\forall r \cdot is\_moregeneral(r, s) \Leftrightarrow is\_moregeneral(r, s'))\end{array}$$

This extended equality can be considered for testing from an unstructured version of UNIFICATION which basically comprises the axioms. The question posed here is whether *EQC* can be used to compute equality for all axioms in a way that test results can be interpreted in terms of correctness. Obviously, *eqc* is coarser than *EQC*. In fact, *EQC* may not be comparable to any finite contextual equality on *substitution* defined in the scope of SUBSTITUTION. As will be shown in this chapter, in order to check all axioms of a specification using a single equality, this equality has to be part of a "compatible" family of equalities, one for each signature in the structure of the specification, where compatible means that the equalities must be comparable in the sight of operations like ***hide*** and ***then***, which may hide/add observers on sorts respectively. The reason is that such test from an unstructured set of axioms does not make any sense if it is not possible to express it in terms of subtests for each subspecification in the structure, since the semantics of these operations is compositional.

Concerning point 2 above, as mentioned there, hidden definitions may impede the checking of interesting axioms. In Example 3.30, *applyl* is hidden in SUBSTITUTION, which means that this operation is not necessarily implemented in the program under test. However, *apply*, which is an exported operation, is mainly defined by axioms in terms of *applyl*. Therefore, this operation can only be suitably tested if an implementation of *applyl* is provided. Given *apply*, it is easy to define *applyl*, but it may be much more complicated to give an implementation of *is_moregeneral*, defined in the UNIFICATION specification.

Regarding test sets (point 3 above), axioms in UNIFICATION may be more effectively checked if particular test sets are considered. For instance, in order to check the unify axiom, test sets may need to include selected pairs of unifiable expressions. On the other hand, axioms in EXPRESSION and SUBSTITUTION may demand test sets composed of different and even random expressions. The question brought up here is whether axioms can be associated with different test sets without raising the oracle problem. More specifically, if a single test set is to be computed from them in order to check all axioms, then what properties should these test sets have in order to allow test results to be interpreted in terms of the semantics of the specification ? □

In the sequel, we use the name *structured testing* to refer to the style of testing which takes the structure of the specification into account in order to test whether an algebra satisfies it. In this case, different tests may be necessary for different parts of the specification. On the other hand, *flat testing* checks an algebra against a flat version of the specification, that is, an unstructured set of visible axioms[1] computed from the original specification. In this case, a single set of tests needs to be performed.

Many advantages arise from performing structured testing rather than flat testing. For instance, more accurate results can be achieved, insofar as there is more flexibility to define approximate equalities and appropriately tackle hidden definitions. In this chapter, it is shown how oracles and test sets can be defined from structured specifications composed of specification-building operations like *union*, *translate* and *hide* [Wirsing, 1990, Hennicker, 1997, Bidoit et al., 1999]. The oracle problem is dealt with in this context, comparing testing results with behavioural satisfaction and considering the instantiation to the specific case of observational satisfaction. Whether structured testing coincides with flat testing is also discussed.

[1] Visible axioms are composed of visible symbols only. Note that hidden symbols need not be implemented and exported by the program under test.

4.2 Family of Equalities

As the aim is to deal with structured specifications, where the signature of different parts of the specification may be different, in the sequel let $\approx = (\approx_\Sigma)_{\Sigma \in Sign}$ be a family of Σ-behavioural equalities, one for each signature Σ, where *Sign* is the category of signatures. A Σ-behavioural equality is defined as before $\approx_\Sigma = (\approx_{\Sigma,A})_{A \in Alg(\Sigma)}$, one partial congruence relation for each algebra A. Whenever Σ is obvious, $\approx$ is used without subscript to denote $\approx_\Sigma$. When A is also obvious, $\approx$ and $\approx_\Sigma$ are used to denote $\approx_{\Sigma,A}$. Likewise, let $\sim = (\sim_\Sigma)_{\Sigma \in Sign}$ and $\simeq = (\simeq_\Sigma)_{\Sigma \in Sign}$ denote families of Σ-approximate equalities. The family $\sim$ is complete (sound) w.r.t $\approx$ if and only if $\forall$ $\Sigma \in Sign$, $\sim_\Sigma$ is complete (sound) w.r.t. $\approx_\Sigma$.

The reduct of a Σ-approximate equality $\sim_\Sigma$ by the morphism $\sigma : \Sigma' \to \Sigma$ considers only the relations of sorts mapped from Σ'.

Definition 4.2 (Reduct) $(\sim_\Sigma)|_\sigma = ((\sim_{\Sigma,A})|_\sigma)_{A \in Alg(\Sigma)}$ *is the* reduct *of* $\sim_\Sigma$ *by* $\sigma : \Sigma' \to \Sigma$, *where* $(\sim_{\Sigma,A})|_\sigma = ((\sim_{\Sigma,A})_{\sigma(s)})_{s \in Sorts(\Sigma')}$.

Concerning signature morphisms and reducts, families of equalities can be compatible, reduction-compatible and translation-compatible as follows.

Definition 4.3 (Compatible) *The family* $\sim$ *is* compatible *with signature morphisms in Sign if for all* $\sigma : \Sigma' \to \Sigma$ *and all* Σ*-algebras* A, $\sim_{\Sigma',A|_\sigma} = (\sim_{\Sigma,A})|_\sigma$.

Proposition 4.4 *The family of literal set-theoretical equality* $=$ *on values of an algebra is compatible.*

Definition 4.5 (Reduction-Compatible) *The family* $\sim$ *is* reduction-compatible *with signature morphisms in Sign if for all* $\sigma : \Sigma' \to \Sigma$ *and all* Σ*-algebras* A, $\sim_{\Sigma',A|_\sigma} \supseteq (\sim_{\Sigma,A})|_\sigma$.

In other words, $\sim$ is reduction-compatible if for all σ, for all A, for all $s \in Sorts(\Sigma')$, for all $v, v' \in A|_\sigma$, $v\ (\sim_{\Sigma,A})_{\sigma(s)}\ v'$ implies $v \sim_{\Sigma',A|_\sigma,s} v'$.

Proposition 4.6 *Let* $Obs(\Sigma)$ *be the set of observable sorts of a signature* Σ. *If* $\forall \Sigma, \Sigma' \in Sign \cdot \forall \sigma : \Sigma' \to \Sigma \cdot \sigma(Obs(\Sigma')) \subseteq Obs(\Sigma)$, *then the family* $(\approx_{\Sigma,Obs(\Sigma)})_{\Sigma \in Sign}$ *of* Σ*-observational equalities is reduction-compatible.*

In order to state Proposition 4.6 above, we assume a fixed assignment of observable sorts to each signature. The assumptions of this proposition seem to be rather strong because of the quantification over all signature morphisms. In practice, one only needs to worry about the ones that actually occur in the specification at hand, and then the condition that observable sorts are preserved is quite natural.

It is easy to check that the family of Σ-observational equalities given in Proposition 4.6 is not compatible. Suppose $\sigma : \Sigma' \hookrightarrow \Sigma$ is an inclusion. In this case, Σ may have more observers than Σ'. Thus, $(\approx_{\Sigma,Obs(\Sigma)})|_\sigma$ may be finer than $\approx_{\Sigma',Obs(\Sigma')}$, that is, $(\approx_{\Sigma,Obs(\Sigma)})|_\sigma \subseteq \approx_{\Sigma',Obs(\Sigma')}$, but $(\approx_{\Sigma,Obs(\Sigma)})|_\sigma \not\supseteq \approx_{\Sigma',Obs(\Sigma')}$.

Definition 4.7 (Translation-Compatible) *The family $\sim$ is* translation-compatible *with signature morphisms in Sign if for all $\sigma : \Sigma' \to \Sigma$ and all Σ-algebras A, $\sim_{\Sigma',A|_\sigma} \subseteq (\sim_{\Sigma,A})|_\sigma$.*

In other words, $\sim$ is translation-compatible if for all σ, for all A, for all $s \in Sorts(\Sigma')$, for all $v, v' \in A|_\sigma$, $v \sim_{\Sigma',A|_\sigma,s} v'$ implies $v\ (\sim_{\Sigma,A})_{\sigma(s)}\ v'$.

In the sequel, let $\Sigma = (S,F)$ and $\Sigma' = (S',F')$ be signatures. Let $\sigma : \Sigma' \to \Sigma$ be a signature morphism. Let $X' = \langle X'_{s'} \rangle_{s' \in S'}$ be an S'-indexed set of variables. Then let $X = \langle X_s \rangle_{s \in S}$ be an S-indexed set of variables such that $X_s = \biguplus_{\sigma(s')=s} X'_{s'}$ for each $s \in S$.[2] The morphism σ gives rise to a translation of Σ'-terms in $T_{\Sigma'}(X')$ to Σ-terms in $T_\Sigma(X)$ and Σ'-formulas to Σ-formulas. If $t' \in T_{\Sigma'}(X')$, then $\sigma(t') \in T_\Sigma(X)$ is obtained by replacing each symbol f in t' by $\sigma(f)$.

Proposition 4.8 *Let A be a Σ-algebra and $\alpha : X \to Dom(\approx_{\Sigma,A})$ and $\alpha' : X' \to Dom(\approx_{\Sigma,A})|_\sigma$ be valuations such that $\forall s' \in S'; x \in X'_{s'} \cdot \alpha'_{s'}(x) = \alpha_{\sigma(s')}(x)$. For any $t \in T_{\Sigma'}(X')$, $\alpha'^{\#}(t) = \alpha^{\#}(\sigma(t))$. In particular, if $t \in T_{\Sigma'}$, ${}^{\#}(t) = {}^{\#}(\sigma(t))$.*

Translations of Σ'-formulas and Σ'-test sets only preserve the testing satisfaction relation under certain conditions on the families of equalities being considered. Let T' be a Σ'-test set and ϕ' be a Σ'-formula.

Theorem 4.9 *Let A be a Σ-algebra and $\alpha : X \to Dom(\approx_{\Sigma,A})$ and $\alpha' : X' \to Dom(\approx_{\Sigma,A})|_\sigma$ be valuations such that $\forall s' \in S'; x \in X'_{s'} \cdot \alpha'_{s'}(x) = \alpha_{\sigma(s')}(x)$. If $\sim$ is reduction-compatible and $\simeq$ is translation-compatible, then $A, \alpha \models^{\sigma(T')}_{\sim_\Sigma, \simeq_\Sigma} \sigma(\phi')$ implies $A|_\sigma, \alpha' \models^{T'}_{\sim_{\Sigma'}, \simeq_{\Sigma'}} \phi'$.*

[2] $\biguplus$ is the disjoint union.

Proof. Let ϕ' be transformed into an equivalent Σ-formula Ψ as in Theorem 3.9 such that for each sub-formula $\neg\psi$ of Ψ, ψ is an equation. This removes all occurrences of $\Leftrightarrow$ and $\Rightarrow$. We proceed by induction on the structure of Ψ.

$(i)\Psi \equiv t = t'$. Suppose $\alpha^{\#}(\sigma(t)) \sim_{\Sigma} \alpha^{\#}(\sigma(t'))$. As $\alpha^{\#}(\sigma(t))$ coincides with $\alpha'^{\#}(t)$ and $\alpha^{\#}(\sigma(t'))$ coincides with $\alpha'^{\#}(t')$ by Proposition 4.8 and because $\sim$ is reduction compatible, that is, $\sim_{\Sigma',A|_{\sigma}} \supseteq (\sim_{\Sigma,A})|_{\sigma}$, then $\alpha'^{\#}(t) \sim_{\Sigma'} \alpha'^{\#}(t')$.

$(ii)\Psi \equiv \neg\psi$. We need to show that $A, \alpha \models^{\sigma(T')}_{\sim_{\Sigma}, \doteqdot_{\Sigma}} \neg(\sigma(\psi))$ implies $A|_{\sigma}, \alpha' \models^{T'}_{\sim_{\Sigma'}, \doteqdot_{\Sigma'}} \neg\psi$, that is, $A, \alpha \models^{\sigma(T')}_{\doteqdot_{\Sigma}, \sim_{\Sigma}} (\sigma(\psi))$ does not hold implies $A|_{\sigma}, \alpha' \models^{T'}_{\doteqdot_{\Sigma'}, \sim_{\Sigma'}} \psi$ does not hold. We show the contrapositive. Suppose $A|_{\sigma}, \alpha' \models^{T'}_{\doteqdot_{\Sigma'}, \sim_{\Sigma'}} \psi$. As ψ is an equation $t = t'$ and $\doteqdot$ is translation-compatible, that is, $\doteqdot_{\Sigma',A|_{\Sigma}} \subseteq (\doteqdot_{\Sigma,A})|_{\sigma}$, and by Proposition 4.8, then $A, \alpha \models^{\sigma(T')}_{\doteqdot_{\Sigma}, \sim_{\Sigma}} \sigma(\psi)$.

$(iii)\Psi \equiv \psi_1 \wedge \psi_2$ **and** $(iv)\Psi \equiv \psi_1 \vee \psi_2$ Trivial as in the proof of Theorem 3.9.

$(v)\Psi \equiv \forall x : s \cdot \psi$. Suppose $A, \alpha \models^{\sigma(T')}_{\sim_{\Sigma}, \doteqdot_{\Sigma}} \sigma(\forall x : s \cdot \psi)$. Then $A, \alpha[x \mapsto v] \models^{\sigma(T')}_{\sim_{\Sigma}, \doteqdot_{\Sigma}} \sigma(\psi)$ for any $v \in {}^{\#}(\sigma(T'))_s$. By induction hypothesis, $A|_{\sigma}, \alpha'[x \mapsto v] \models^{T'}_{\sim_{\Sigma'}, \doteqdot_{\Sigma'}} \psi$. So, $A|_{\sigma}, \alpha'[x \mapsto v] \models^{T'}_{\sim_{\Sigma'}, \doteqdot_{\Sigma'}} \psi$ for all $v \in {}^{\#}(\sigma(T'))_s$. By Proposition 4.8, ${}^{\#}(\sigma(T'))$ coincides with ${}^{\#}(T')$. Thus, $A|_{\sigma}, \alpha' \models^{T'}_{\sim_{\Sigma'}, \doteqdot_{\Sigma'}} \forall x : s \cdot \psi$.

$(vi)\Psi \equiv \exists x : s \cdot \psi$. Similar to item (v).

□

Theorem 4.10 *Let A be a Σ-algebra and $\alpha : X \to Dom(\approx_{\Sigma,A})$ and $\alpha' : X' \to Dom(\approx_{\Sigma,A})|_{\sigma}$ be valuations such that $\forall s' \in S'; x \in X'_{s'} \cdot \alpha'_{s'}(x) = \alpha_{\sigma(s')}(x)$. If $\sim$ is translation-compatible and $\doteqdot$ is reduction-compatible, then $A|_{\sigma}, \alpha' \models^{T'}_{\sim_{\Sigma'}, \doteqdot_{\Sigma'}} \phi'$ implies $A, \alpha \models^{\sigma(T')}_{\sim_{\Sigma}, \doteqdot_{\Sigma}} \sigma(\phi')$.*

Proof. Similar to the proof of Theorem 4.9. □

Obviously, if $\sim$ and $\doteqdot$ are compatible, the implication holds in both directions, that is, translations preserve the testing satisfaction relation.

Corollary 4.11 *Let A be a Σ-algebra and $\alpha : X \to Dom(\approx_{\Sigma,A})$ and $\alpha' : X' \to Dom(\approx_{\Sigma,A})|_{\sigma}$ be valuations such that $\forall s' \in S'; x \in X'_{s'} \cdot \alpha'_{s'}(x) = \alpha_{\sigma(s')}(x)$. If $\sim$ and $\doteqdot$ are compatible, then $A, \alpha \models^{\sigma(T')}_{\sim_{\Sigma}, \doteqdot_{\Sigma}} \sigma(\phi')$ if and only if $A|_{\sigma}, \alpha' \models^{T'}_{\sim_{\Sigma'}, \doteqdot_{\Sigma'}} \phi'$.*

Proof. Directly from Theorems 4.9 and 4.10. □

Finally, as one may suspect, translations of Σ'-formulas and Σ'-test sets preserve the behavioural satisfaction relation if the family of behavioural equalities is compatible.

Theorem 4.12 *Let A be a Σ-algebra and $\alpha : X \to Dom(\approx_{\Sigma,A})$ and $\alpha' : X' \to Dom(\approx_{\Sigma,A})|_\sigma$ be valuations such that $\forall s' \in S'; x \in X'_{s'} \cdot \alpha'_{s'}(x) = \alpha_{\sigma(s')}(x)$. If $\approx$ is compatible, then $A, \alpha \models_{\approx_\Sigma} \sigma(\phi')$ if and only if $A|_\sigma, \alpha' \models_{\approx_{\Sigma'}} \phi'$.*

Proof. Follows from Definition 4.3 and Proposition 4.8. □

4.3 Basic Specifications with Testing Interface

With regard to specification-based testing, test sets are usually defined in terms of specification concepts rather than programs [Stepney, 1995, Donat, 1997, Le Gall, 1999]. Providing test sets in the specification interface means that they can be uniformly and more effectively defined as specifications are created. This makes it possible to deliver programs with the necessary tools for checking them. Motivations for this come also from the notion of *deliverables* [Burstall and Mckinna, 1992], which consist of a program together with a proof of correctness. For testing, the intention is that for each program, an oracle, defined from specification axioms and given families of equalities, together with specific test sets, can be provided. Specifications with test sets are called *specifications with testing interface*. Basic specifications with testing interface are defined as follows.

Definition 4.13 (Basic Specification) *Let $\Sigma = \langle S, F \rangle$ be a signature, T be a Σ-test set and Φ be a set of Σ-sentences. A* basic *(flat) specification with testing interface is defined as follows.*

Syntax: $SP = \langle \Sigma, \Phi, T \rangle$

Semantics:

- $Sig(SP) = \Sigma$
- $Test(SP) = T$

- $Mod_{\approx}(SP) = \{A \in Alg(\Sigma) \mid A \models_{\approx_\Sigma} \Phi\}$
- $ChMod_{\sim,\doteq}(SP) = \{A \in Alg(\Sigma) \mid A \models^{T}_{\sim_\Sigma,\doteq_\Sigma} \Phi\}$

where $Mod_{\approx}(SP)$ denotes the class of real models *of SP w.r.t. a Σ-behavioural equality $\approx_\Sigma = (\approx_{\Sigma,A})_{A\in Alg(\Sigma)}$ and $ChMod_{\sim,\doteq}(SP)$ denotes the class of* checkable models *of SP by testing w.r.t. the Σ-approximate equalities $\sim_\Sigma = (\sim_{\Sigma,A})_{A\in Alg(\Sigma)}$, $\doteq_\Sigma = (\doteq_{\Sigma,A})_{A\in Alg(\Sigma)}$ and the Σ-test set T. Behavioural satisfaction is chosen instead of standard satisfaction for defining the class of real models.*

The following theorems show some situations in which a real model A is also a checkable model and vice-versa. Essentially, they are consequences of the main results obtained in Chapter 3 for flat specifications in Theorems 3.9, 3.10, 3.21 and 3.22.

Theorem 4.14 *Let $SP = \langle \Sigma, \Phi, T\rangle$ be a basic specification. If $\sim_\Sigma$ is a complete equality, $\doteq_\Sigma$ is a sound equality, and for all $\phi \in \Phi$, ϕ has only positive occurrences of $\forall$ and negative occurrences of $\exists$, then $A \in Mod_{\approx}(SP)$ implies $A \in ChMod_{\sim,\doteq}(SP)$.*

Proof. Show that $A \models_{\approx} \phi$ implies $A \models^{T}_{\sim,\doteq} \phi$ for all $\phi \in \Phi$. By Theorem 3.9. □

Theorem 4.14 covers a common use of $\forall$ and $\exists$ whereby, as one might expect, incorrect programs can be accepted by testing satisfaction, whereas Theorem 4.15 below, the dual of Theorem 4.14, covers a case where correct programs can be rejected by testing, but success in testing guarantees correctness.

Theorem 4.15 *Let $SP = \langle \Sigma, \Phi, T\rangle$ be a basic specification. If $\sim_\Sigma$ is a sound equality, $\doteq_\Sigma$ is a complete equality, and for all $\phi \in \Phi$, ϕ has only negative occurrences of $\forall$ and positive occurrences of $\exists$, then $A \in ChMod_{\sim,\doteq}(SP)$ implies $A \in Mod_{\approx}(SP)$.*

Proof. By Theorem 3.10. □

The conditions on Theorem 4.15 mean that it is rarely applicable. Furthermore, as mentioned before, assumptions on quantifiers can always be dropped in Theorem 4.14 if T is unbiased and in Theorem 4.15 if T is valid (see Section 3.4).

Theorem 4.16 *Let $SP = \langle \Sigma, \Phi, T\rangle$ be a basic specification. If $\sim_\Sigma$ is a complete equality, $\doteq_\Sigma$ is a sound equality and for all $\phi \in \Phi$, T is unbiased for $A, \sim, \doteq, \phi$, then $A \in Mod_{\approx}(SP)$ implies $A \in ChMod_{\sim,\doteq}(SP)$.*

Proof. By Theorem 3.21. □

Theorem 4.17 *Let $SP = \langle \Sigma, \Phi, T \rangle$ be a basic specification. If $\sim_\Sigma$ is a sound equality, $\doteqdot_\Sigma$ is a complete equality and for all $\phi \in \Phi$, T is valid for $A, \sim, \doteqdot, \phi$, then $A \in ChMod_{\sim,\doteqdot}(SP)$ implies $A \in Mod_{\approx}(SP)$.*

Proof. By Theorem 3.22. □

4.4 Structured Specifications with Testing Interface

The following definition presents some specification-building operations [Wirsing, 1990, Hennicker, 1997] for creating structured specifications with testing interface.

Definition 4.18 (Structured Specification)

$$\begin{array}{rcl} SP & = & \langle \Sigma, \Phi, T \rangle \mid \\ & & SP_1 \cup SP_2 \mid \\ & & \textsf{translate } SP' \textsf{ with } \sigma \mid \\ & & \textsf{hide sorts } S' \textsf{ opns } F' \textsf{ in } SP' \end{array}$$

where $\langle \Sigma, \Phi, T \rangle$ is a basic specification, SP', SP_1 and SP_2 are structured specifications with $Sig(SP_1) = Sig(SP_2)$, σ is a signature morphism, S' is a set of sorts and F' is a set of function declarations.

The set of operations chosen above corresponds to a small set of primitive operations which enable individual problems found when testing from structured specifications to be analysed in isolation. These operations can be combined in order to define more complex and interesting ones found in the literature [Wirsing, 1990, Hennicker, 1997], like *enrichment* and *arbitrary union* or sum of specifications. However, as one might expect, new problems do not arise from these combinations. This explains why they are not included in Definition 4.18.

Let $\sigma : \Sigma' \hookrightarrow \Sigma$ be an inclusion. Enrichment, represented by ***then*** in CASL, can be defined in terms of *translate* and *union*:

$$SP' \text{ then sorts } S \text{ opns } F \text{ axioms } \Phi \text{ test set } T \stackrel{\text{def}}{=} \langle \Sigma, \Phi, T\rangle \cup \text{translate } SP' \text{ with } \sigma$$

where SP' is a structured specification, S is a set of sorts, F is a set of function declarations, $\Sigma = Sig(SP') \cup (S, F)$, T is a Σ-test set and Φ is a set of Σ-sentences. The arbitrary union of specifications, with possibly different signatures, represented by ***and*** in CASL, can be expressed as:

$$SP_1 \text{ and } SP_2 \stackrel{\text{def}}{=} \text{translate } SP_1 \text{ with } \sigma_1 \cup \text{translate } SP_2 \text{ with } \sigma_2$$

where $\sigma_1 : Sig(SP_1) \hookrightarrow \Sigma$, $\sigma_2 : Sig(SP_2) \hookrightarrow \Sigma$ and $\Sigma = Sig(SP_1) \cup Sig(SP_2)$.

Furthermore, when $\sigma : \Sigma \hookrightarrow \Sigma'$ is an inclusion, a clear-cut case of the *derive* operation can be defined in terms of *hide*:

$$\text{derive from } SP' \text{ by } \sigma \stackrel{\text{def}}{=} \text{hide sorts } S' \text{ opns } F' \text{ in } SP'$$

where $\Sigma' = Sig(SP')$, $\Sigma = \Sigma' - \langle S', F'\rangle$.

Instantiation of generic specifications can be defined in terms of *union* and *translate* in the usual way. For example, see [Sannella et al., 1992].

Example* 4.19** In Example 3.30, EXPRESSION is defined by adding new symbols and axioms to the arbitrary union of specifications, and then hiding some auxiliary functions. This specification is defined in terms of ***and, ***then*** and ***hide***, while SUBSTITUTION and UNIFICATION are defined in terms of ***then*** and ***hide***. □

The testing satisfaction relation presented in Definition 3.8 cannot always check all axioms of each part of a structured specification because they may refer to hidden sorts and functions not implemented in the algebra under test. Therefore, it may be necessary to look at visible (testable) axioms only.

Definition 4.20 (Testable Axioms) *The set of testable (visible) axioms of a structured specification SP can be defined as follows.*

1. $TAx(\langle \Sigma, \Phi, T\rangle) \stackrel{def}{=} \Phi$
2. $TAx(SP_1 \cup SP_2) \stackrel{def}{=} TAx(SP_1) \cup TAx(SP_2)$

3. $TAx(\textsf{translate } SP' \textsf{ with } \sigma) \stackrel{def}{=} \sigma(TAx(SP'))$

4. $TAx(\textsf{hide sorts } S' \textsf{ opns } F' \textsf{ in } SP') \stackrel{def}{=} TAx(SP') \cap Sen(\Sigma)$, *where* $\Sigma = Sig(\textsf{hide sorts } S' \textsf{ opns } F' \textsf{ in } SP')$

It can be noticed that name clashes do not occur in $TAx(SP)$ because axioms with hidden symbols are completely excluded.

***Example* 4.21** Revisiting Example 3.30 again, the testable axioms of EXPRESSION are the ones which do not refer to *variablesl* and *is_validl*, that is, only the first, second, sixth and seventh axioms are testable, without requiring an implementation of hidden operations. □

In the next subsections, the semantics of the specification-building operations from Definition 4.18 is given as usual. Also, some lemmas concerning whether flat testing, which is represented by testing satisfaction of visible axioms ($A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$), implies structured testing, which is represented by membership in the class of checkable models ($A \in ChMod_{\sim,\doteq}(SP)$), and vice-versa. These lemmas are used to prove the main theorems presented in Section 4.5. Example 4.22 below is used to illustrate the points discussed in the sequel.

***Example* 4.22** This example presents a specification with testing interface of sorted lists. Let Σ_1 and Σ_2 be signatures defined as follows.

$$\Sigma_1 = \left[\begin{array}{ll} \textsf{sorts} & \{int, list\} \\ \textsf{opns} & \{nil : list \\ & \;cons : int * list \to list \\ & \;head : list \to int \\ & \;tail : list \to list\} \end{array}\right]$$

$$\Sigma_2 = \left[\begin{array}{ll} \textsf{sorts} & \{int, bool, list\} \\ \textsf{opns} & \{\ldots\ \Sigma_1\ operations\ \ldots \\ & \;is\_sorted : list \to bool \\ & \;sort : list \to list\} \end{array}\right]$$

Let $\sigma : \Sigma_1 \hookrightarrow \Sigma_2$ be a signature morphism. The following are well-formed structured specifications.

$$List = \left[\begin{array}{ll} \textsf{sig} & \Sigma_1 \\ \textsf{axioms} & \forall l : list; x : int \cdot cons(x,l) \neq l \\ & \forall l : list; x : int \cdot head(cons(x,l)) = x \\ & \forall l : list; x : int \cdot tail(cons(x,l)) = l \\ \textsf{test set} & T_1 = (T_{1s})_{s \in \{int, list\}} \end{array}\right]$$

$$SList1 = \textsf{translate } List \textsf{ by } \sigma$$

$$SList2 = \left[\begin{array}{ll} \textsf{sig} & \Sigma_2 \\ \textsf{axioms} & is\_sorted(nil) = true \\ & \ldots \\ & \forall l : list \cdot is\_sorted(sort(l)) = true \\ \textsf{test set} & T_2 = (T_{2s})_{s \in \{int, bool, list\}} \end{array}\right]$$

$$SList = SList1 \cup SList2$$

$$IntList = \textsf{hide sorts } \emptyset \textsf{ opns } \{is\_sorted\} \textsf{ in } SList$$

where T_1 and T_2 are arbitrary Σ_1 and Σ_2-test sets respectively. □

4.4.1 Union

The union of two specifications with the same signature gives a new specification whose models are the ones in the intersection of the sets of models of the constituent specifications.

Definition 4.23 (*Union*) *Let SP_1 and SP_2 be structured specifications, such that their signatures coincide, that is, $Sig(SP_1) = Sig(SP_2)$. Then $SP = SP_1 \cup SP_2$ is defined as follows.*

- $Sig(SP) \stackrel{def}{=} Sig(SP_1) = Sig(SP_2)$
- $Test(SP) \stackrel{def}{=} Test(SP_1) \cup Test(SP_2)$
- $Mod_{\approx}(SP) \stackrel{def}{=} Mod_{\approx}(SP_1) \cap Mod_{\approx}(SP_2)$
- $ChMod_{\sim,\doteq}(SP) \stackrel{def}{=} ChMod_{\sim,\doteq}(SP_1) \cap ChMod_{\sim,\doteq}(SP_2)$

Let $SP = SP_1 \cup SP_2$. Then $A \in ChMod_{\sim,\doteq}(SP)$ may not imply $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$, even if both SP_1 and SP_2 are basic specifications, because axioms of SP_1 and SP_2 are tested with a different, possibly bigger, test set $Test(SP)$ in the latter. Similarly, the converse may not hold. The reason is that, due to the possible presence of existential quantifiers, the result obtained by checking an axiom with $Test(SP)$ may not imply the result obtained by checking the same axiom with $Test(SP_1) \subseteq Test(SP)$. In Example 4.22, T_1 and T_2 may be different, so the axioms of $SList1$ and $SList2$ may be checked with different test sets when testing $SList$ in the former and latter way. Thus, it is necessary to specify some restrictions on test sets. One possible restriction is that they must coincide. In Example 4.22, this implies that T_1 and T_2 must be syntactically equal, and then T_2 cannot include ground terms with symbols not in Σ_1.

Lemma 4.24 *Let $SP = SP_1 \cup SP_2$ be a structured specification and A be a $Sig(SP)$-algebra. If*

1. (a) *For all $s \in sorts(Sig(SP))$, $Test(SP_1)_s = Test(SP_2)_s$ and*
   (b) *$A \in ChMod_{\sim,\doteq}(SP_1)$ implies $A \models^{Test(SP_1)}_{\sim,\doteq} TAx(SP_1)$ and*
   (c) *$A \in ChMod_{\sim,\doteq}(SP_2)$ implies $A \models^{Test(SP_2)}_{\sim,\doteq} TAx(SP_2)$*

   then $A \in ChMod_{\sim,\doteq}(SP)$ implies $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$

2. (a) *For all $s \in sorts(Sig(SP))$, $Test(SP_1)_s = Test(SP_2)_s$ and*
   (b) *$A \models^{Test(SP_1)}_{\sim,\doteq} TAx(SP_1)$ implies $A \in ChMod_{\sim,\doteq}(SP_1)$ and*
   (c) *$A \models^{Test(SP_2)}_{\sim,\doteq} TAx(SP_2)$ implies $A \in ChMod_{\sim,\doteq}(SP_2)$*

   then $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$ implies $A \in ChMod_{\sim,\doteq}(SP)$

Proof.

1. Suppose $A \in ChMod_{\sim,\doteq}(SP)$. Then $A \in ChMod_{\sim,\doteq}(SP_1)$ and $A \in ChMod_{\sim,\doteq}(SP_2)$ by Definition 4.23. By assumption $1(b)$, $A \models^{Test(SP_1)}_{\sim,\doteq} TAx(SP_1)$ and by assumption $1(c)$, $A \models^{Test(SP_2)}_{\sim,\doteq} TAx(SP_2)$. Then, by $1(a)$, $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP_1)$ and $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP_2)$. So, $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$.

2. Suppose $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$. As $TAx(SP_1) \subseteq TAx(SP)$ and $TAx(SP_2) \subseteq TAx(SP)$, then $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP_1)$ and $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP_2)$. By assumption $2(a)$, $A \models^{Test(SP_1)}_{\sim,\doteq} TAx(SP_1)$ and $A \models^{Test(SP_2)}_{\sim,\doteq} TAx(SP_2)$. By assumption $2(b)$, $A \in ChMod_{\sim,\doteq}(SP_1)$ and by assumption $2(c)$ $A \in ChMod_{\sim,\doteq}(SP_2)$. Hence, by Definition 4.23, $A \in ChMod_{\sim,\doteq}(SP)$.

□

Lemma 4.24 suggests that if $Test(SP_1)$ and $Test(SP_2)$ are different for $s \in S$, then it might be interesting to substitute one for the other. If the axioms of SP have only positive $\forall$ and negative $\exists$ (negative $\forall$ and positive $\exists$), supersets of $Test(SP_1)$ ($Test(SP_2)$) are always unbiased (valid). So, it is better to opt for the bigger one if that is possible (see Theorem 3.20). In Example 4.22, if $T_1 \subseteq T_2$ and T_2 is defined in the scope of Σ_1, then T_2 can replace T_1. Otherwise, T_2 has to be trimmed to fit in the scope of Σ_1.

Assumption (a) in Lemma 4.24 is a strong one indeed. This is further discussed in Section 4.5. One way of weakening this condition is to make assumptions on quantifiers. From Theorem 3.17, any test set is unbiased if there are only positive occurrences of $\forall$ and negative occurrences of $\exists$ in the set of axioms to be checked (see also Theorem 3.18).

Lemma 4.25 *Let $SP = SP_1 \cup SP_2$ be a structured specification and A be a $Sig(SP)$-algebra. If*

1. *(a) Axioms of SP have only negative $\forall$ and positive $\exists$ and*
   (b) $A \in ChMod_{\sim,\simeq}(SP_1)$ implies $A \models^{Test(SP_1)}_{\sim,\simeq} TAx(SP_1)$ and
   (c) $A \in ChMod_{\sim,\simeq}(SP_2)$ implies $A \models^{Test(SP_2)}_{\sim,\simeq} TAx(SP_2)$

   then $A \in ChMod_{\sim,\simeq}(SP)$ implies $A \models^{Test(SP)}_{\sim,\simeq} TAx(SP)$

2. *(a) Axioms of SP have only positive $\forall$ and negative $\exists$ and*
   (b) $A \models^{Test(SP_1)}_{\sim,\simeq} TAx(SP_1)$ implies $A \in ChMod_{\sim,\simeq}(SP_1)$ and
   (c) $A \models^{Test(SP_2)}_{\sim,\simeq} TAx(SP_2)$ implies $A \in ChMod_{\sim,\simeq}(SP_2)$

   then $A \models^{Test(SP)}_{\sim,\simeq} TAx(SP)$ implies $A \in ChMod_{\sim,\simeq}(SP)$

Proof. Similar to the proof of Lemma 4.24. □

It is important to remark that assumptions $1(a)$ and $2(a)$ in Lemma 4.25 are contrary to the ones in Theorems 4.14 and 4.15 respectively, according to the direction of implication we are interested in. Thus, Lemma 4.25 may not be very useful when comparing flat testing to structured testing.

Note that, by Definition 4.13, assumptions $1(b)$-(c) and $2(b)$-(c) in Lemma 4.24 will hold if SP_1 and SP_2 are basic specifications.

4.4.2 Translate

Given a specification and a signature morphism, *translate* produces a new specification from the former according to the morphism.

Definition 4.26 (*Translate*) *Let SP' be a structured specification. SP =* translate *SP'* with σ *is defined as follows, where $\sigma : \Sigma' \to \Sigma$ is a signature morphism and $Sig(SP') = \Sigma'$.*

- $Sig(SP) \stackrel{def}{=} \Sigma$
- $Test(SP) \stackrel{def}{=} \sigma(Test(SP'))$
- $Mod_{\approx}(SP) \stackrel{def}{=} \{A \in Alg(\Sigma) \mid A|_{\sigma} \in Mod_{\approx}(SP')\}$
- $ChMod_{\sim,\simeq}(SP) \stackrel{def}{=} \{A \in Alg(\Sigma) \mid A|_{\sigma} \in ChMod_{\sim,\simeq}(SP')\}$

Let SP = translate SP' with $\sigma : \Sigma' \to \Sigma$. Then $A \in ChMod_{\sim,\simeq}(SP)$ implies $A \models^{Test(SP)}_{\sim,\simeq} TAx(SP)$ and vice-versa, provided that this holds in both directions for SP' and appropriate assumptions on equalities are made. These assumptions are required since translation of terms and formulas does not always respect the testing satisfaction relation. Consider Example 4.22 and let $\sim$ denote a family of approximate equalities defined by a finite set of observable contexts. Clearly, $\sim_{\Sigma_2}$ can be finer than $\sim_{\Sigma_1}$, since Σ_2 has an additional $list$ observer (is_sorted). Moreover, it is easy to check that flat testing of $SList1$ considers $\sim_{\Sigma_2}$, whereas structured testing considers $\sim_{\Sigma_1}$. Therefore, compatibility conditions are necessary.

Lemma 4.27 *Let SP =* translate *SP'* with σ. *Let A be a Σ-algebra. If*

1. *(a)* $\sim$ *is* translation-compatible *and* $\simeq$ *is* reduction-compatible *and*

   (b) $A|_{\sigma} \in ChMod_{\sim,\simeq}(SP')$ *implies* $A|_{\sigma} \models^{Test(SP')}_{\sim_{\Sigma'},\simeq_{\Sigma'}} TAx(SP')$

   then $A \in ChMod_{\sim,\simeq}(SP)$ *implies* $A \models^{Test(SP)}_{\sim_{\Sigma},\simeq_{\Sigma}} TAx(SP)$.

2. *(a)* $\sim$ *is* reduction-compatible *and* $\simeq$ *is* translation-compatible *and*

   (b) $A|_{\sigma} \models^{Test(SP')}_{\sim_{\Sigma'},\simeq_{\Sigma'}} TAx(SP')$ *implies* $A|_{\sigma} \in ChMod_{\sim,\simeq}(SP')$

   then $A \models^{Test(SP)}_{\sim_{\Sigma},\simeq_{\Sigma}} TAx(SP)$ *implies* $A \in ChMod_{\sim,\simeq}(SP)$.

Proof.

1. Suppose $A \in ChMod_{\sim,\simeq}(SP)$. Then $A|_{\sigma} \in ChMod_{\sim,\simeq}(SP')$ by Definition 4.26. By assumption, $A|_{\sigma} \models^{Test(SP')}_{\sim_{\Sigma'},\simeq_{\Sigma'}} TAx(SP')$. Thus, by Theorem 4.10, $A \models^{Test(SP)}_{\sim_{\Sigma},\simeq_{\Sigma}} TAx(SP)$.

2. Suppose $A \models^{Test(SP)}_{\sim_\Sigma,\simeq_\Sigma} TAx(SP)$. Then $A|_\sigma \models^{Test(SP')}_{\sim_{\Sigma'},\simeq_{\Sigma'}} TAx(SP')$ by Theorem 4.9. By assumption, $A|_\sigma \in ChMod_{\sim,\simeq}(SP')$. Thus, $A \in ChMod_{\sim,\simeq}(SP)$, by Definition 4.26.

$\square$

Assumptions $1(b)$ and $2(b)$ in Lemma 4.27 will hold if SP' is a basic specification.

4.4.3 Hide

The *hide* operation constructs a new specification from *SP* whose models are the models of *SP* with some sorts (and their respective test sets) and functions removed.

Definition 4.28 (*Hide*) *Let SP' be a specification. SP =* hide sorts *S'* opns *F'* in *SP' is defined as follows.*

- $Sig(SP) \stackrel{def}{=} Sig(SP') - \langle S', F' \rangle$
- $Test(SP) \stackrel{def}{=} Test(SP') \cap T_{Sig(SP)}$
- $Mod_{\approx}(SP) \stackrel{def}{=} \{A'|_\Sigma \mid A' \in Mod_{\approx}(SP')\}$
- $ChMod_{\sim,\simeq}(SP) \stackrel{def}{=} \{A'|_\Sigma \mid A' \in ChMod_{\sim,\simeq}(SP')\}$

where S' is a set of sorts, F' is a set of function declarations and $\Sigma = Sig(SP)$ is required to be a well-formed signature.

Let SP = hide sorts S' opns F' in SP'. In order to check whether $A \in ChMod_{\sim,\simeq}(SP)$, it is necessary to check whether $A' \in ChMod_{\sim,\simeq}(SP')$ for some $Sig(SP')$-algebra A' such that $A = A'|_{Sig(SP)}$. In Example 4.22, one must provide a *SList*-algebra A' (with an implementation of *is_sorted*) in order to check A against *IntList* by structured testing. Clearly, $A \models^{Test(SP)}_{\sim,\simeq} TAx(SP)$ may not imply $A \in ChMod_{\sim,\simeq}(SP)$ as SP' may have axioms not in $TAx(SP)$ and the test set of *SP* may be smaller than the test set of SP'. Likewise, the converse may not hold. Note that $TAx(IntList)$ does not include any axiom referring to *is_sorted*. Furthermore, when *hide* is present, compatibility assumptions on equalities are also necessary in order to compare flat and structured testing, since different signatures are considered when testing in one or the other style.

Lemma 4.29 *Let SP =* hide sorts *S'* opns *F'* in *SP' with $\Sigma' = Sig(SP')$ and $\Sigma = Sig(SP)$. Let A' be a Σ'-algebra. If*

1. *(a)* $\sim$ *is* reduction-compatible *and* $\simeq$ *is* translation-compatible *and*

   (b) Axioms of SP have only positive $\forall$ *and negative* $\exists$ *and*

   (c) $A' \in ChMod_{\sim,\simeq}(SP')$ *implies* $A' \models^{Test(SP')}_{\sim_{\Sigma'},\simeq_{\Sigma'}} TAx(SP')$

   then $A'|_\Sigma \in ChMod_{\sim,\simeq}(SP)$ *implies* $A'|_\Sigma \models^{Test(SP)}_{\sim_\Sigma,\simeq_\Sigma} TAx(SP)$

2. *(a)* $TAx(SP) = TAx(SP')$ *and*

   (b) $\sim$ *is* translation-compatible *and* $\simeq$ *is* reduction-compatible *and*

   (c) Axioms of SP have only negative $\forall$ *and positive* $\exists$ *and*

   (d) $A' \models^{Test(SP')}_{\sim_{\Sigma'},\simeq_{\Sigma'}} TAx(SP')$ *implies* $A' \in ChMod_{\sim,\simeq}(SP')$

   then $A'|_\Sigma \models^{Test(SP)}_{\sim_\Sigma,\simeq_\Sigma} TAx(SP)$ *implies* $A'|_\Sigma \in ChMod_{\sim,\simeq}(SP)$

Proof.

1. Suppose $A'|_\Sigma \in ChMod_{\sim,\simeq}(SP)$. Then $A' \in ChMod_{\sim,\simeq}(SP')$ by Definition 4.28. By assumption $1(c)$, $A' \models^{Test(SP')}_{\sim_{\Sigma'},\simeq_{\Sigma'}} TAx(SP')$. By definition of $TAx(SP)$, $A' \models^{Test(SP')}_{\sim_{\Sigma'},\simeq_{\Sigma'}} TAx(SP)$. Because $Test(SP) \subseteq Test(SP')$, by assumption $1(b)$ and Theorem 3.20, $A' \models^{Test(SP)}_{\sim_{\Sigma'},\simeq_{\Sigma'}} TAx(SP)$. Let $\sigma : \Sigma \hookrightarrow \Sigma'$ be the inclusion between Σ and Σ' ($\Sigma \subseteq \Sigma'$). Then $A' \models^{\sigma(Test(SP))}_{\sim_{\Sigma'},\simeq_{\Sigma'}} \sigma(TAx(SP))$, as $\sigma(Test(SP)) = Test(SP)$ and $\sigma(TAx(SP)) = TAx(SP)$. By assumption $1(a)$ and Theorem 4.9, $A'|_\Sigma \models^{Test(SP)}_{\sim_\Sigma,\simeq_\Sigma} TAx(SP)$.

2. Suppose $A'|_\Sigma \models^{Test(SP)}_{\sim_\Sigma,\simeq_\Sigma} TAx(SP)$. Then $A' \models^{\sigma(Test(SP))}_{\sim_{\Sigma'},\simeq_{\Sigma'}} \sigma(TAx(SP))$ by assumption $2(b)$ and Theorem 4.10, that is, $A' \models^{Test(SP)}_{\sim_{\Sigma'},\simeq_{\Sigma'}} TAx(SP)$, where $\sigma : \Sigma \hookrightarrow \Sigma'$. By $2(a)$, $A' \models^{Test(SP)}_{\sim_{\Sigma'},\simeq_{\Sigma'}} TAx(SP')$. Because $Test(SP) \subseteq Test(SP')$, by assumption $2(c)$ and Theorem 3.20, $A' \models^{Test(SP')}_{\sim_{\Sigma'},\simeq_{\Sigma'}} TAx(SP')$. By assumption $2(d)$, $A' \in ChMod_{\sim,\simeq}(SP')$. Hence, by Definition 4.28, $A'|_{\Sigma'} \in ChMod_{\sim,\simeq}(SP)$.

$\square$

Assumption $2(a)$ in Lemma 4.29, whereby SP' cannot have axioms with symbols in S' and F', is a strong restriction. However, one can see this assumption as requiring that specifications should be transformed such that hidden axioms are replaced by their visible consequences.

Contrary to Lemma 4.25, assumptions $1(b)$ and $2(c)$ in Lemma 4.29 are perfectly compatible with the ones in Theorems 4.14 and 4.15 respectively. Nevertheless, as in Lemma 4.24, these assumptions on quantifiers can also be replaced as follows.

Lemma 4.30 *Let* SP = hide sorts S' opns F' in SP' *with* $\Sigma' = Sig(SP')$ *and* $\Sigma = Sig(SP)$. *Let* A' *be a* Σ'*-algebra. If*

1. (a) $\sim$ *is* reduction-compatible *and* $\doteqdot$ *is* translation-compatible *and*

   (b) $Test(SP)_s = Test(SP')_s$, *for all* $s \in sorts(Sig(SP))$ *and*

   (c) $A' \in ChMod_{\sim,\doteqdot}(SP')$ *implies* $A' \models^{Test(SP')}_{\sim_{\Sigma'},\doteqdot_{\Sigma'}} TAx(SP')$

   then $A'|_\Sigma \in ChMod_{\sim,\doteqdot}(SP)$ *implies* $A'|_\Sigma \models^{Test(SP)}_{\sim_\Sigma,\doteqdot_\Sigma} TAx(SP)$

2. (a) $TAx(SP) = TAx(SP')$ *and*

   (b) $\sim$ *is* translation-compatible *and* $\doteqdot$ *is* reduction-compatible *and*

   (c) $Test(SP)_s = Test(SP')_s$, *for all* $s \in sorts(Sig(SP))$ *and*

   (d) $A' \models^{Test(SP')}_{\sim_{\Sigma'},\doteqdot_{\Sigma'}} TAx(SP')$ *implies* $A' \in ChMod_{\sim,\doteqdot}(SP')$

   then $A'|_\Sigma \models^{Test(SP)}_{\sim_\Sigma,\doteqdot_\Sigma} TAx(SP)$ *implies* $A'|_\Sigma \in ChMod_{\sim,\doteqdot}(SP)$

Proof. The proof is similar to the proof of Lemma 4.29. □

This implies that hidden symbols cannot be used to define any tests in the structure of *SP*, which is again very restrictive.

4.4.4 Terminology

Some terminology of structured specifications with testing interface is presented in this section. Due to the fact that test sets are part of the specification, one could think of defining valid/unbiased specifications by analogy with valid/unbiased test sets.

Definition 4.31 (Valid and Unbiased Specifications) *Let* SP *be a structured specification and* A *be a* Σ*-algebra.* SP *is* unbiased (valid) *for* $A,\sim,\doteqdot$ *if and only if*

$SP \equiv \langle \Sigma, \Phi, T\rangle$**.** T *is unbiased (valid) for* A, $\sim_\Sigma$, $\doteqdot_\Sigma$, Φ*;*

$SP \equiv SP_1 \cup SP_2$**.** SP_1 *is unbiased (valid) for* A, $\sim$, $\doteqdot$ *and* SP_2 *is unbiased (valid) for* A, $\sim$, $\doteqdot$*;*

$SP \equiv$ translate SP' with σ**.** SP' *is unbiased (valid) for* $A|_\sigma$, $\sim$, $\doteqdot$*;*

$SP \equiv$ hide sorts S' opns F' in SP'**.** *There exists a* $Sig(SP')$*-algebra* A' *with* $A = A'|_{Sig(SP)}$ *such that* SP' *is unbiased (valid) for* A', $\sim$, $\doteqdot$.

It is easy to check that, owing to the *hide* operation, *SP* is unbiased (valid) for A, $\sim$, $\hat{=}$ does not imply $Test(SP)$ is unbiased (valid) for A, $\sim$, $\hat{=}$, $TAx(SP)$. The reason is that the former may consider test sets with values defined from hidden sorts. The converse does not hold either.

Interpretation of the results of flat testing cannot always be given for structured specifications. Concerning test sets, assumption (a) in Lemma 4.24 has to be met at each occurrence of *union* together with assumptions 1(b) and 2(c) in either Lemma 4.29 or 4.30 at each occurrence of *hide*, depending on which direction of the implication one is interested in. To illustrate the need for the former assumption and some problems it brings, consider the following example.

***Example* 4.32** Suppose Σ_2 and $SList2$ in Example 4.22 are modified as follows.

$$\Sigma_2 = \left[\begin{array}{ll} \mathsf{sorts} & \{int, bool, nat, list\} \\ \mathsf{opns} & \{\ldots\ \Sigma_1\ operations\ \ldots \\ & is\_sorted : list \rightarrow bool \\ & sort : list \rightarrow list \\ & take : nat * list \rightarrow list \\ & len : list \rightarrow nat \\ & concat : list * list \rightarrow list\} \end{array}\right]$$

where the $take$, len and $concat$ functions are added. $take(n,l)$ returns a list containing the first n items of l. The function $concat$ returns the concatenation of two lists. The function len returns the length of a list.

$$SList2 = \left[\begin{array}{ll} \mathsf{sig} & \Sigma_2 \\ \mathsf{axioms} & is\_sorted(nil) = true \\ & \ldots \\ & \forall l : list \cdot is\_sorted(sort(l)) = true \\ & \forall n : nat; l,k : list \cdot take(n,l) = k \Leftrightarrow \\ & \qquad\qquad \exists s : list \cdot concat(k,s) = l \wedge len(k) = n \\ & \ldots \\ \mathsf{test\ set} & T_2 = (T_{2s})_{s \in \{int, bool, nat, list\}} \end{array}\right]$$

where axioms on $take$ are added.

Because nat is not in Σ_1, then by Definition 4.26, $Test(SList1)_{nat} = \sigma(T_1)_{nat} = \emptyset$. In order to fulfil assumption 1(a) in Lemma 4.24, so that the results of flat testing can be interpreted, $Test(SList_2)_{nat}$ must be empty, and then axioms on *take* cannot be checked. □

According to the example above, assumption (a) in Lemma 4.24 is too strong. For instance, all specifications defined by the *then* operation where a new sort is introduced would have the problem mentioned above (see the SUBSTITUTION specification and the *substitution* sort in Example 3.30). One way of weakening this assumption is by ignoring sorts which have no test set associated with them, at least, in the structure of one of the specifications in the union. Notice that there are no test sets explicitly associated with *nat* and no axioms on this sort in the structure of $SList1$. Thus, this sort does not raise the problem regarding testing with different test sets by flat and structured testing which is discussed in Subsection 4.4.1. Then, in the sequel, specifications are only required to be consistent w.r.t to test sets. But before giving this definition, we need to define an auxiliary notion.

Consider the following signature morphism $\sigma : \Sigma' \rightarrow \Sigma$ which can be used to rename specifications. Note that the signature of *translate* SP' by σ (Σ in this case), with $\Sigma' = Sig(SP')$, may have sorts that do not correspond to any sort defined in Σ'. In example 4.32, *nat* is not in the signature of *List*, even though this sort is in the signature of $SList1 =$ translate *List* by σ. In this case, *nat* is said to be *innocuous* in $SList1$. If a sort s is *innocuous* in a specification SP, this sort belong to the signature of SP, but there is no test set explicitly associated with it in this specification. Also, no axioms refer to it. Only *translate* can introduce *innocuous* sorts in the structure of a specification. And only the union of SP_1 with another specification SP_2 in which a given sort s is not *innocuous*, can make this sort s which is *innocuous* in SP_1 become not *innocuous* in the resulting specification $SP_1 \cup SP_2$. For instance, *nat* is not *innocuous* in $SList2$, and, consequently, it is not *innocuous* in *SList* and *IntList*.

Definition 4.33 (Innocuous sort) *Let $s \in sorts(Sig(SP))$. Whether s is* innocuous *in SP is defined as follows.*

1. *s is never* innocuous *in* $\langle \Sigma, \Phi, T \rangle$.

2. *s is* innocuous *in $SP_1 \cup SP_2$ if and only if s is* innocuous *in SP_1 and s is* innocuous *in SP_2.*

3. *s is* innocuous *in* translate SP' with σ *if and only if either*

   - $\neg(\exists s' : sorts(Sig(SP')) \cdot \sigma(s') = s)$ *or*

- $\exists s' : sorts(Sig(SP')) \cdot \sigma(s') = s$ *and* s' *is* innocuous *in* SP'.

4. s *is* innocuous *in* hide sorts S' opns F' in SP' *if and only if* s *is* innocuous *in* SP'.

Consistent specifications require test sets to coincide at each occurrence of *union*, unless a given sort is *innocuous* in at least one of the subspecifications which composes the union.

Definition 4.34 (Consistent Specification) *A structured specification SP is* consistent w.r.t test sets *if, for every subspecification* $SP_1 \cup SP_2$ *of SP, for all* $s \in Sig(SP_1 \cup SP_2)$*, either:*

- $Test(SP_1)_s = Test(SP_2)_s$ *or*

- s *is* innocuous *in* SP_1 *or* s *is* innocuous *in* SP_2.

***Example* 4.35** In Example 4.32, *IntList* is consistent w.r.t. test sets if T_1 coincides with T_2 for all sorts in Σ_2 except for *nat* and *bool*, since they are *innocuous* in *SList*1. Therefore, no test set is associated with these sorts. On the contrary, *nat* and *bool* are associated with test set $(T_2)_{\{bool,nat\}}$ in *SList*2. □

The following is a corollary of Lemma 4.24.

Corollary 4.36 *Let* $SP = SP_1 \cup SP_2$ *be consistent w.r.t. test sets and A be a* $Sig(SP)$*-algebra. If*

1. (a) $A \in ChMod_{\sim,\doteq}(SP_1)$ *implies* $A \models^{Test(SP_1)}_{\sim,\doteq} TAx(SP_1)$ *and*

   (b) $A \in ChMod_{\sim,\doteq}(SP_2)$ *implies* $A \models^{Test(SP_2)}_{\sim,\doteq} TAx(SP_2)$

   then $A \in ChMod_{\sim,\doteq}(SP)$ *implies* $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$

2. (a) $A \models^{Test(SP_1)}_{\sim,\doteq} TAx(SP_1)$ *implies* $A \in ChMod_{\sim,\doteq}(SP_1)$ *and*

   (b) $A \models^{Test(SP_2)}_{\sim,\doteq} TAx(SP_2)$ *implies* $A \in ChMod_{\sim,\doteq}(SP_2)$

   then $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$ *implies* $A \in ChMod_{\sim,\doteq}(SP)$

It may be possible to transform a given specification into a consistent specification w.r.t test sets by substituting the bigger test sets for the smaller ones (see Theorem 3.20). But, not all specifications can be transformed into a consistent one in this way. In Example 4.22, if $T_1 \subseteq T_2$, T_2 must refer to symbols in Σ_1 only. In Chapter 5, a normal form is presented to produce specifications that are consistent w.r.t. test sets. For this, specifications may be required to be compatible w.r.t test sets as follows.

Definition 4.37 (Compatible Specification) *A structured specification SP is* compatible w.r.t. test sets *if for all $\sigma : \Sigma' \to \Sigma''$ arising in SP, for all subspecifications SP' and SP'' of SP with $\Sigma' = Sig(SP')$ and $\Sigma'' = Sig(SP'')$, $Test(SP'') \subseteq \sigma(T_{\Sigma'})$, that is, $Test(SP'')$ is completely defined by symbols mapped from Σ'.*

Results on compatible specifications are presented in Section 5.2. The intention is to replace the consistency condition on test sets. In Example 4.32, $IntList$ is compatible w.r.t. test sets if T_2 does not include terms defined from $sort$, $take$, len, $concat$ and is_sorted. Obviously, not all specifications which are consistent w.r.t test sets are also compatible and vice-versa.

***Example* 4.38** Let $SList2$ in Example 4.22 be redefined as follows.

$$SList2' = \left[\begin{array}{ll} \textsf{sig} & \Sigma_2 \cup \{reverse : list \to list\} \\ \textsf{axioms} & is\_sorted(nil) = true \\ & \ldots \\ & \forall l \cdot is\_sorted(sort(l)) = true \\ & \ldots \\ & reverse(nil) = nil \\ & \ldots \\ \textsf{test set} & T_2 = (T_{2s})_{s \in \{int, bool, nat, list\}} \end{array}\right]$$

$$SList2 = \textsf{hide sorts } \emptyset \textsf{ opns } \{reverse\} \textsf{ in } SList2'$$

$IntList$ is consistent w.r.t test sets if T_1 coincides (except for nat and $bool$) with $T_2 \cap T_{\Sigma_2}$, that is, T_2 restricted to the Σ_2 signature. But, in this case, $IntList$ may not be compatible w.r.t. test sets, because T_2 may include terms referring to $reverse$. On the other hand, $InsList$ is compatible w.r.t. test sets if both T_1 and T_2 are defined in the scope of Σ_1. But, because these test sets may differ, $IntList$ may not be consistent w.r.t. test sets. □

Regarding test sets and *hide*, in order to meet assumptions $1(b)$ and $2(c)$ in Lemma 4.30 and handle the problem raised by *hide*, specifications may be required to be visible w.r.t test sets in the following sense.

Definition 4.39 (Visible Specification) *A structured specification SP is* visible w.r.t test sets *if for every subspecification $SP' = \textsf{hide sorts } S \textsf{ opns } F \textsf{ in } SP''$, $Test(SP')_s = Test(SP'')_s$, for all $s \in sorts(Sig(SP'))$, that is, hidden symbols are not used to define test sets in the structure of SP.*

***Example* 4.40** Consider again Example 4.38. The $IntList$ specification is visible w.r.t. test sets if $Test(SList2')_{list} = Test(SList2)_{list}$ and $Test(IntList)_{bool} = Test(SList)_{bool}$, that is, neither is_sorted nor $reverse$ is referred to by terms in T_2. □

The following is a corollary of Lemma 4.30.

Corollary 4.41 *Let SP =* hide sorts *S'* opns *F'* in *SP' be visible w.r.t. test sets, where $\Sigma' = Sig(SP')$ and $\Sigma = Sig(SP)$. Let A' be a Σ'-algebra. If*

1. *(a) $\sim$ is* reduction-compatible *and $\doteq$ is* translation-compatible *and*

   (b) $A' \in ChMod_{\sim,\doteq}(SP')$ implies $A' \models^{Test(SP')}_{\sim_{\Sigma'},\doteq_{\Sigma'}} TAx(SP')$

   then $A'|_\Sigma \in ChMod_{\sim,\doteq}(SP)$ implies $A'|_\Sigma \models^{Test(SP)}_{\sim_\Sigma,\doteq_\Sigma} TAx(SP)$

2. *(a) $TAx(SP) = TAx(SP')$ and*

   (b) $\sim$ is translation-compatible *and $\doteq$ is* reduction-compatible *and*

   (c) $A' \models^{Test(SP')}_{\sim_{\Sigma'},\doteq_{\Sigma'}} TAx(SP')$ implies $A' \in ChMod_{\sim,\doteq}(SP')$

   then $A'|_\Sigma \models^{Test(SP)}_{\sim_\Sigma,\doteq_\Sigma} TAx(SP)$ implies $A'|_\Sigma \in ChMod_{\sim,\doteq}(SP)$

It is easy to check that if SP is compatible w.r.t test sets then SP is visible w.r.t test sets, but the converse may not hold (see Definitions 4.37 and 4.28). Also, if SP is consistent and visible w.r.t test sets, then SP is compatible w.r.t. test sets. But, obviously, the converse may not hold, as the latter does not take union into account.

Furthermore, in order to meet assumption 2(a) in Lemma 4.29, specifications may be required to be transparent as follows.

Definition 4.42 (Transparent Specification) *Let SP be a structured specification. SP is* transparent *if it has no occurrences of* hide *in its structure.*

The following is a corollary of Lemma 4.29.2.

Corollary 4.43 *Let SP =* hide sorts *S'* opns *F'* in *SP' be transparent with $\Sigma' = Sig(SP')$ and $\Sigma = Sig(SP)$. Let A' be a Σ'-algebra. If*

1. *(a) $\sim$ is* translation-compatible *and $\doteq$ is* reduction-compatible *and*

   (b) Axioms of SP have only negative $\forall$ and positive $\exists$ and

   (c) $A' \models^{Test(SP')}_{\sim_{\Sigma'},\doteq_{\Sigma'}} TAx(SP')$ implies $A' \in ChMod_{\sim,\doteq}(SP')$

   then $A'|_\Sigma \models^{Test(SP)}_{\sim_\Sigma,\doteq_\Sigma} TAx(SP)$ implies $A'|_\Sigma \in ChMod_{\sim,\doteq}(SP)$

If hidden axioms of a specification can be replaced by their visible consequences in such a way that hidden symbols can be discarded, this specification can be turned into a transparent one.

4.5 Structured Testing versus Flat Testing

In this section, the main results of this chapter are presented by comparing structured testing and flat testing and explaining them in terms of correctness. *Structured testing* of a Σ-algebra A against a structured specification SP corresponds to membership in the class of checkable models of SP, that is, $A \in ChMod_{\sim,\doteq}(SP)$, whereas *flat testing* corresponds to the testing satisfaction of visible axioms of SP, that is, $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$. The former is based on the structure of SP and may consist of more than one test with specific test sets applied for specific groups of axioms (see e.g. Definition 4.23) and may demand additional implementation of sorts and operations not in A, due to hiding. On the other hand, flat testing is a monolithic experiment based on an unstructured view of the specification without considering hidden elements and using a single test set $Test(SP)$ which is deduced from SP and applied to all axioms in $TAx(SP)$.

***Example* 4.44** Let A be an $IntList$-algebra. In Example 4.22, flat testing from $IntList$ corresponds to the test:

$$A \models^{Test(IntList)}_{\sim_\Sigma,\doteq_\Sigma} TAx(IntList)$$

where $\Sigma = Sig(IntList)$, $TAx(IntList)$ is the set of visible axioms of $IntList$ according to Definition 4.20 and $Test(IntList)$ is the test set of $IntList$ which is restricted to terms composed of visible symbols. On the other hand, from Definitions 4.23, 4.26 and 4.28, structured testing, that is, $A \in ChMod_{\sim,\doteq}(IntList)$, corresponds to the following tests:

$$A'|_{\Sigma_1} \models^{T_1}_{\sim_{\Sigma_1},\doteq_{\Sigma_1}} \Phi_1 \;\; and \;\; A' \models^{T_2}_{\sim_{\Sigma_2},\doteq_{\Sigma_2}} \Phi_2$$

where Φ_1 and Φ_2 are the axioms of $List$ and $SList2$ respectively and A' is a $SList$-algebra with $A = A'|_{IntList}$. Note that the first test is almost the same as $A \models^{T_1}_{\sim_{\Sigma_1},\doteq_{\Sigma_1}} \Phi_1$. □

Advantages of structured versus flat testing are presented throughout this section. In the next subsections, some theorems are presented which relate structured/flat testing to a notion of correctness w.r.t. the class of real models. Let SP be a structured specification with $\Sigma = Sig(SP)$ and A be a Σ-algebra.

4.5.1 Structured Testing and Correctness

Theorem 4.45 is a generalisation of Theorem 4.14 for structured specifications. Under certain conditions incorrect programs can be accept by structured testing, that is, not every checkable model is a real model, but any real model is a checkable model. The assumptions that families of equalities need to be complete (sound) seem to be strong, but only signatures arising in the structure of *SP* need to be considered when defining these families.

Theorem 4.45 *If $\sim$ is complete, $\doteq$ is sound, and the axioms of SP have only positive occurrences of $\forall$ and negative occurrences of $\exists$, then $A \in Mod_{\approx}(SP)$ implies $A \in ChMod_{\sim,\doteq}(SP)$.*

Proof. The proof is conducted by induction on the structure of *SP*.

(i) $SP = \langle \Sigma, \Phi, T \rangle$**.** From Theorem 4.14.

(ii) $SP = SP_1 \cup SP_2$**.** Suppose $A \in Mod_{\approx}(SP)$. Then $A \in Mod_{\approx}(SP_1)$ and $A \in Mod_{\approx}(SP_2)$. By induction hypothesis, $A \in ChMod_{\sim,\doteq}(SP_1)$ and $A \in ChMod_{\sim,\doteq}(SP_2)$. Thus, $A \in ChMod_{\sim,\doteq}(SP)$.

(iii) $SP =$ translate SP' with σ, with $\sigma : \Sigma' \to \Sigma$**.** Suppose $A \in Mod_{\approx}(SP)$. Then, $A|_{\sigma} \in Mod_{\approx}(SP')$ by Definition 4.26. By induction hypothesis, $A|_{\sigma} \in ChMod_{\sim,\doteq}(SP')$. Thus, $A \in ChMod_{\sim,\doteq}(SP)$.

(iv) $SP =$ hide sorts S' opns F' in SP'**.** Suppose $A \in Mod_{\approx}(SP)$. Then there exists a $Sig(SP')$-algebra A' such that $A' \in Mod_{\approx}(SP')$ and $A = A'|_{Sig(SP)}$ by Definition 4.28. By induction hypothesis, $A' \in ChMod_{\sim,\doteq}(SP')$. Hence, $A \in ChMod_{\sim,\doteq}(SP)$.

□

Theorem 4.46 below is a generalisation of Theorem 4.15 for structured specifications and the dual of Theorem 4.45. This theorem shows that under the conditions stated, real models can be rejected by structured testing, but any checkable model is a real model.

Theorem 4.46 *If $\sim$ is sound, $\doteq$ is complete and the axioms of SP have only negative occurrences of $\forall$ and positive occurrences of $\exists$, then $A \in ChMod_{\sim,\doteq}(SP)$ implies $A \in Mod_{\approx}(SP)$.*

Example 4.22 matches Theorem 4.45 since the axioms of *IntList* have only positive $\forall$ and negative $\exists$. Given $\sim$ complete and $\simeq$ sound, any *IntList*-algebra A can be checked against *IntList* by structured testing in the sense that an interpretation can be given for test results based on Theorem 4.45, no matter how T_1 and T_2 are defined. In this case, only the signatures Σ_1 and Σ_2 in the families $\sim$ and $\simeq$ need to be considered. In practice, an oracle for *IntList* can be defined as a procedure which drives the tests required. Note that hidden axioms are taken into account and an implementation of is_sorted is needed.

Once more, assumptions on quantifiers can be dropped if specifications are unbiased and/or valid.

Theorem 4.47 *If $\sim$ is complete, $\simeq$ is sound and SP is unbiased for $A,\sim,\simeq$, then $A \in Mod_{\approx}(SP)$ implies $A \in ChMod_{\sim,\simeq}(SP)$.*

Proof. The proof is conducted by induction on the structure of *SP* and is similar to the proof of Theorem 4.45, except that the basic case follows from Theorem 4.16 instead of 4.14. □

Theorem 4.48 *If $\simeq$ is sound, $\sim$ is complete and SP is valid for $A,\simeq,\sim$, then $A \in ChMod_{\simeq,\sim}(SP)$ implies $A \in Mod_{\approx}(SP)$.*

Proof. Similar to the proof of Theorem 4.47. □

Interesting properties of specifications may be expressed by hidden axioms built from visible and hidden symbols (see *sort* in Example 4.22) and these axioms cannot always be replaced by their visible consequences. According to Definition 4.28, in order to check hidden axioms by structured testing, an implementation of hidden symbols may be required for each occurrence of hide, which is a reasonable constructive requirement. In a similar way to structured testing, compositional proof systems for implementations of structured specifications may require a persistent extension SP_i' of SP_i to be checked against SP' in order to prove that SP_i is an implementation of hide sorts S' opns F' in SP' [Hennicker, 1997]. Also, [Farrés-Casals, 1990] shows that under those circumstances the visible part of the specification can be proved by importing the hidden part into the implementation. One drawback of testing hidden axioms is that the implementation of hidden symbols may itself have errors which might lead to wrong conclusions regarding correctness.

However, the main goal of testing is to uncover errors so that a debugger can be used to pinpoint them. Effort to provide an implementation of hidden symbols which conforms to their specification is compensated by the fact that this can be used to check different implementations of the visible part. Moreover, it is likely that these hidden symbols represent standard components and functions already implemented and verified. Another drawback is that the specification of hidden symbols might not have an implementation (see Example 1 in [Kahrs and Sannella, 1998]). In this case, unless the specification can be transformed, only flat testing can be applied.

4.5.2 Structured Testing and Flat Testing

If *SP* is not a basic specification, structured testing does not always coincide with flat testing. However, the following theorems show special cases where either one or the other direction of the implication may hold.

Theorem 4.49 *Let SP be consistent w.r.t. test sets. If $\sim$ and $\doteq$ are compatible and the axioms of SP have only positive occurrences of $\forall$ and negative occurrences of $\exists$, then $A \in ChMod_{\sim,\doteq}(SP)$ implies $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$.*

Proof. By induction on *SP*. From Definition 4.13, Corollary 4.36(1), Lemmas 4.27(1) and 4.29(1). □

Notice that assumptions on quantifiers in Theorem 4.49 are due to the *hide* operation which may restrict test sets to be composed of visible symbols only. Undoubtedly, these assumptions can be dropped by requiring *SP* to be visible w.r.t test sets. Theorem 4.50 below, the dual of 4.49, also requires *SP* to be transparent.

Theorem 4.50 *Let SP be transparent and consistent w.r.t. test sets. If $\sim$ and $\doteq$ are compatible and the axioms of SP have only negative occurrences of $\forall$ and positive occurrences of $\exists$, then $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$ implies $A \in ChMod_{\sim,\doteq}(SP)$.*

Proof. By induction on *SP*. From Definition 4.13, Corollaries 4.36(2) and 4.43, Lemma 4.27(2). □

Equivalence of structured and flat testing requires specifications to be consistent w.r.t. test sets and transparent, families of equalities to be compatible and also contradictory assumptions on quantifiers, unless specifications are visible w.r.t test sets. According to Theorem 4.49, in Example 4.22, success in structured testing implies success in flat testing if T_1 and T_2 coincide and the families $\sim$ and $\simeq$ are compatible at least for Σ_1, Σ_2 and $Sig(IntList)$. Obviously, Theorem 4.50 cannot be used, since $IntList$ has positive $\forall$ and is not transparent.

4.5.3 Flat Testing and Correctness

The definition of real models relies on the compositional semantics of specifications. Therefore, in contrast to structured testing, further assumptions on specifications and equalities are required to identify whether a real model is accepted by flat testing and vice-versa.

Theorem 4.51 *Let SP be consistent w.r.t. test sets. If $\sim$ is complete and compatible and $\simeq$ is sound and compatible and the axioms of SP have only positive occurrences of $\forall$ and negative occurrences of $\exists$, then $A \in Mod_{\approx}(SP)$ implies $A \models^{Test(SP)}_{\sim,\simeq} TAx(SP)$.*

Proof. Follows by Theorems 4.45 and 4.49. □

Again to obtain a result from flat testing in Example 4.22, according to Theorem 4.51, no great flexibility is given to define T_1 and T_2, since $IntList$ must be consistent and also equalities on $Sig(IntList)$ must belong to compatible families. For the consistency requirement, $(T_2)_{List}$ cannot have any term that involves $sort$. This even forbids tests like $head(cons(x, sort(l))) = x$ to be performed, where the $sort$ operation is somehow exercised. Thus, also because hidden axioms cannot be considered and $sort$ is defined in terms of is_sorted, the operation $sort$ cannot be checked by flat testing. Structured testing is the only way of effectively verifying this operation.

Theorem 4.52 *Let SP be consistent w.r.t. test sets and transparent. If $\sim$ is sound and compatible and $\simeq$ is complete and compatible and the axioms of SP have only negative occurrences of $\forall$ and positive occurrences of $\exists$, then $A \models^{Test(SP)}_{\sim,\simeq} TAx(SP)$ implies $A \in Mod_{\approx}(SP)$.*

Proof. Follows by Theorems 4.50 and 4.46. □

Assumptions on quantifiers in Theorem 4.51 can only be replaced by unbiased conditions if specifications are also transparent. Although not actually checked by flat testing, hidden definitions have to be considered when comparing testing satisfaction and membership in the class of real models. Suppose we assume that SP is unbiased for $A, \sim, \simeq$. In this case, if SP = hide sorts S' opns F' in SP', then $Test(SP)_s \subseteq$

$Test(SP')_s$, for all $s \in sorts(SP)$. Thus, *SP* has to be transparent, unless assumptions on quantifiers are made (see Theorem 4.51).

Theorem 4.53 *Let SP be consistent w.r.t. test sets and transparent. If $\sim$ is complete and compatible and $\doteq$ is sound and compatible and SP is unbiased for $A, \sim, \doteq$, then $A \in Mod_{\approx}(SP)$ implies $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$.*

Proof. By induction on the structure of *SP*.

(i) $SP = \langle \Sigma, \Phi, T \rangle$**.** From Theorem 4.16, since $A \in ChMod_{\sim,\doteq}(SP)$ is equivalent to $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$.

(ii) $SP = SP_1 \cup SP_2$**.** Suppose $A \in Mod_{\approx}(SP)$. Then $A \in Mod_{\approx}(SP_1)$ and $A \in Mod_{\approx}(SP_2)$. Since *SP* is consistent w.r.t test sets, by induction hypothesis, $A \models^{Test(SP_1)}_{\sim,\doteq} TAx(SP_1)$ and $A \models^{Test(SP_1)}_{\sim,\doteq} TAx(SP_1)$. Thus, because *SP* is consistent w.r.t test sets, $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$.

(iii) SP = translate SP' with σ**, with** $\sigma : \Sigma' \to \Sigma$**.** Suppose $A \in Mod_{\approx}(SP)$. Then $A|_\sigma \in Mod_{\approx}(SP')$. By induction hypothesis, $A|_\sigma \models^{Test(SP')}_{\sim,\doteq} TAx(SP')$. Hence, by Theorem 4.10, $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$.

□

In Theorem 4.54, the dual of Theorem 4.53, *SP* is required to be valid for $A, \sim, \doteq$ and transparent as follows.

Theorem 4.54 *Let SP be consistent w.r.t. test sets and transparent. If $\sim$ is complete and compatible and $\doteq$ is sound and compatible, and SP is valid for $A, \sim, \doteq$, then $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$ implies $A \in Mod_{\approx}(SP)$.*

Proof. Similar to the proof of Theorem 4.53. □

Due to the hide operation, in order to check all axioms of a structured specification *SP* in a non-compositional way, this specification may need to be normalised by a process similar to normalisation of ordinary specifications [Bidoit et al., 1999] to give a specification $nf(SP) =$ hide sorts S' opns F' in $\langle \Sigma', \Phi', T' \rangle$, which is equivalent to *SP* w.r.t. signature equivalence and model class equality. If such a normal form $nf(SP)$ can be computed, in order to test A against $nf(SP)$ (and consequently, *SP*), it would be necessary to test $A + \Delta(A)$ against $\langle \Sigma', \Phi', T' \rangle$, where $\Delta(A)$ implements the hidden sorts and functions. It is easy to check that whenever the assumptions of

Theorem 4.52 hold, $A+\Delta(A) \models^{T'}_{\sim,\doteq} \Phi'$ implies $A+\Delta(A) \in Mod_{\approx}(\langle \Sigma', \Phi', T' \rangle)$, and then, by Definition 4.28, $A \in Mod_{\approx}(nf(SP))$. Normalisation of specifications with testing interface is the subject of Chapter 5.

Finally, Figure 4.1 illustrates the main results obtained in this section when comparing structured testing, flat testing and correctness.

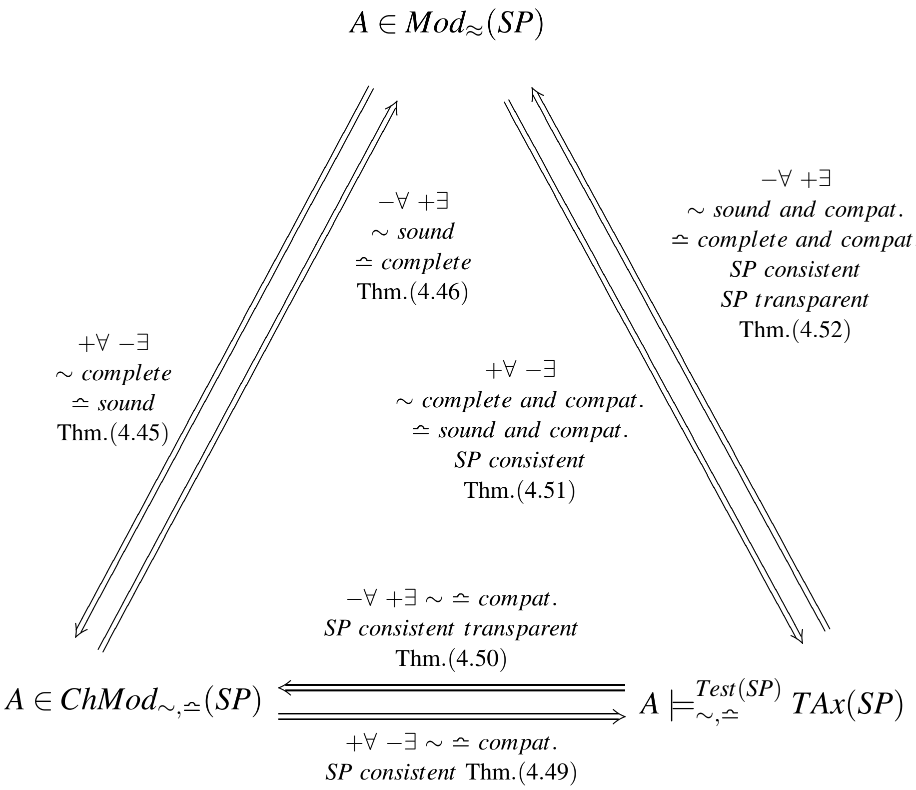

Figure 4.1: Structured testing, flat testing and correctness.

4.6 The Observational Case

Let $Obs(\Sigma)$ be the set of observable sorts of a signature Σ and assume that $\forall \Sigma, \Sigma' \in Sign \cdot \forall \sigma : \Sigma' \to \Sigma \cdot \sigma(Obs(\Sigma')) \subseteq Obs(\Sigma)$. When restricting behavioural equalities to be observational equalities, a family $\approx = (\approx_{\Sigma,Obs(\Sigma)})_{\Sigma \in Sign}$ of observational equalities can be defined, where $\approx_{\Sigma,Obs(\Sigma)} = (\approx_{\Sigma,Obs(\Sigma),A})_{A \in Alg(\Sigma)}$ is a Σ-observational equality with one observational equality for each Σ-algebra A. This section is focused on observational equalities and how families of Σ-approximate equalities can be defined so that the theorems of Section 4.5 hold. By Proposition 4.6, this family of observational equalities is reduction-compatible.

4.6.1 Structured Testing

For structured testing, it is necessary to define a complete and a sound family of approximate equalities w.r.t. the family of observational equalities. Let $C_{Obs(\Sigma)}$ be the set of all Σ-contexts of observable sorts in $Obs(\Sigma)$. Let $\overline{\sim} = (\overline{\sim}_{\Sigma,\mathcal{C}})_{\Sigma \in Sign, \mathcal{C} \subseteq C_{Obs(\Sigma)}}$ be a family of contextual equalities, for example, built from the set $\mathcal{C}$ of crucial contexts as in Section 2.4. It is easy to check that the family $\overline{\sim}$ is complete w.r.t. $\approx$. In Example 4.22, given that *list* is a non-observable sort and *int* and *bool* are observable sorts, Σ_1 has $head(z)$ as crucial context, whereas Σ_2 has $head(z)$ and $is\_sorted(z)$. Furthermore, let $\simeq = (\simeq_\Sigma)_{\Sigma \in Sign}$ be the family of literal set-theoretical equalities on the values of the Σ-algebras. This family is clearly sound w.r.t. $\approx$. The following is a corollary of Theorems 4.45 and 4.46.

Corollary 4.55 *Let*

- $\approx = (\approx_{\Sigma,Obs(\Sigma)})_{\Sigma \in Sign}$ *and*
- $\overline{\sim} = (\overline{\sim}_{\Sigma,\mathcal{C}})_{\Sigma \in Sign, \mathcal{C} \subseteq C_{Obs(\Sigma)}}$ *be any family of* Σ*-contextual equalities and*
- $\simeq = (\simeq_\Sigma)_{\Sigma \in Sign}$ *be the family of literal equalities on values of the algebras.*

Then

1. *If the axioms of SP have only positive occurrences of* $\forall$ *and negative occurrences of* $\exists$*, then* $A \in Mod_{\approx}(SP)$ *implies* $A \in ChMod_{\overline{\sim},\simeq}(SP)$.
2. *If the axioms of SP have only negative occurrences of* $\forall$ *and positive occurrences of* $\exists$*, then* $A \in ChMod_{\simeq,\overline{\sim}}(SP)$ *implies* $A \in Mod_{\approx}(SP)$.

4.6.2 Flat Testing

For flat testing, from Theorems 4.51 and 4.52, it is necessary to define *compatible* families of equalities. It is easy to check that $\overline{\sim}$ (from previous subsection) is not translation-compatible for the same reasons as $\approx$ is not. In Example 4.22, this is due to the observer *is_sorted* which is added in Σ_2. Let *SP preserves encapsulation*, that is, for all $\sigma : \Sigma' \to \Sigma$ arising in *SP*, Σ does not introduce new observations for corresponding sorts in Σ'. In Example 4.22, this means that *is_sorted* has to be defined in Σ_1 and Σ_2 and cannot be hidden. Clearly, if this restriction is applied, both $\approx$ and $\overline{\sim}$ are always compatible. By Proposition 4.4, $\simeq$ is compatible. The following is a corollary of Theorems 4.51 and 4.52.

Corollary 4.56 *Let*

- *SP be consistent w.r.t test sets and preserve encapsulation and*
- $\approx = (\approx_{\Sigma,Obs(\Sigma)})_{\Sigma \in Sign}$ *and*
- $\overline{\sim} = (\overline{\sim}_{\Sigma,\mathcal{C}})_{\Sigma \in Sign, \mathcal{C} \subseteq \mathcal{C}_{Obs(\Sigma)}}$ *be any family of* Σ*-contextual equalities and*
- $\simeq = (\simeq_\Sigma)_{\Sigma \in Sign}$ *be the family of literal equalities on values of the algebras.*

Then

1. *If the axioms of SP have only positive occurrences of* $\forall$ *and negative occurrences of* $\exists$*, then* $A \in Mod_{\approx}(SP)$ *implies* $A \models^{Test(SP)}_{\overline{\sim},\simeq} TAx(SP)$.
2. *If SP is transparent and its axioms have only negative occurrences of* $\forall$ *and positive occurrences of* $\exists$*, then* $A \models^{Test(SP)}_{\simeq,\overline{\sim}} TAx(SP)$ *implies* $A \in Mod_{\approx}(SP)$.

It practice, even if *SP* does not preserve encapsulation, it is possible to define a compatible family $\overline{\sim}$ by focusing on the signatures which arise in *SP* and determining a single set of contexts $\mathcal{C}$ for each non-observable sort.

4.7 Application

In this section, structured and flat testing from specifications with testing interface are compared through an example in CASL. The intention is to furnish a more complete explanation of the points underlined in Sections 4.5 and 4.6 and assess structured and flat testing.

For the sake of simplicity, specification-building operations like *then* and *and* (commented on at the beginning of Section 4.4) are not decomposed into the corresponding combination of *union* and *translate*. Thus, the semantics of *then* and *and* (obtained by expanding the definitions of these operations) are presented below. Given a specification, a set of sorts and function declarations, a set of axioms, and a test set, *then* produces a new specification whose models are the ones of the former which also satisfy the new axioms with the new test set.

Definition 4.57 (Then) *Let* SP' *be a structured specification,* S *be a set of sorts,* F *be a set of operations,* $\Sigma = Sig(SP') \cup (S,F)$*,* T *be a* Σ*-test set and* Φ *be a set of* Σ*-sentences. Then* $SP = SP'$ then sorts S opns F axioms Φ test set T *is defined as follows.*

- $Sig(SP) \stackrel{def}{=} \Sigma$
- $Test(SP) \stackrel{def}{=} Test(SP') \cup T$
- $Mod_{\approx}(SP) \stackrel{def}{=} \{A \in Alg(\Sigma) \mid A|_{Sig(SP')} \in Mod_{\approx}(SP') \text{ and } A \models_{\approx} \Phi\}$
- $ChMod_{\sim,\simeq}(SP) \stackrel{def}{=} \{A \in Alg(\Sigma) \mid A|_{Sig(SP')} \in ChMod_{\sim,\simeq}(SP') \text{ and } A \models^{T}_{\sim,\simeq} \Phi\}$

The arbitrary union or sum of two specifications with *distinct* signatures gives a new specification whose models are the ones that, when reduced to the corresponding signatures, are models of both former specifications.

Definition 4.58 (And) *Let SP_1 and SP_2 be structured specifications with $\Sigma_1 = Sig(SP_1)$ and $\Sigma_2 = Sig(SP_2)$. Let $\Sigma = \Sigma_1 \cup \Sigma_2$. Then $SP = SP_1$* and *SP_2 is defined as follows.*

- $Sig(SP) \stackrel{def}{=} \Sigma$
- $Test(SP) \stackrel{def}{=} Test(SP_1) \cup Test(SP_2)$
- $Mod_{\approx}(SP) \stackrel{def}{=} \{A \in Alg(\Sigma) \mid A|_{\Sigma_1} \in Mod_{\approx}(SP_1) \text{ and } A|_{\Sigma_2} \in Mod_{\approx}(SP_2)\}$
- $ChMod_{\sim,\simeq}(SP) \stackrel{def}{=} \{A \in Alg(\Sigma) \mid A|_{\Sigma_1} \in ChMod_{\sim,\simeq}(SP_1) \text{ and } A|_{\Sigma_2} \in ChMod_{\sim,\simeq}(SP_2)\}$

Definition 4.20 can then be extended to the following.

Definition 4.59 (Testable Axioms (*extended*)) *The set of testable (visible) axioms of a structured specification SP can be defined as follows.*

1. $TAx(\langle \Sigma, \Phi, T \rangle) \stackrel{def}{=} \Phi$
2. $TAx(SP_1 \cup SP_2) \stackrel{def}{=} TAx(SP_1) \cup TAx(SP_2)$
3. $TAx(\text{translate } SP' \text{ with } \sigma) \stackrel{def}{=} \sigma(TAx(SP'))$
4. $TAx(\text{hide sorts } S' \text{ opns } F' \text{ in } SP') \stackrel{def}{=} TAx(SP') \cap Sen(\Sigma)$, *where* $\Sigma = Sig(\text{hide sorts } S' \text{ opns } F' \text{ in } SP')$
5. $TAx(SP' \text{ then sorts } S \text{ opns } F \text{ axioms } \Phi \text{ test set } T) \stackrel{def}{=} \Phi \cup TAx(SP')$
6. $TAx(SP_1 \text{ and } SP_2) \stackrel{def}{=} TAx(SP_1) \cup TAx(SP_2)$

Obviously, all theorems in Sections 4.5 and 4.6 hold if both *then* and *and* are taken into account. Moreover, since these operations can be defined in terms of *union* and *translate*, they inherit the problems faced when both *union* and *translate* are considered.

The next sections draw attention again to the UNIFICATION specification presented in Example 3.30, showing how this can be extended to a specification with testing interface and how test oracles can be planned for performing structured and flat testing. The advantages and limitations of structured and flat testing are highlighted.

4.7.1 Specifying the Testing Interface

In general, specification languages like CASL do not support specifications with testing interface, that is, they do not provide special constructions for permitting test sets to be defined as part of the specification. Thus, it is not possible here to give real specifications with testing interface. However, it is possible, at least, to conceive how this could be done in languages with such support. As any other specification symbol that is exported, the testing interface is mapped into program values. Note that $Test(SP) = (Test(SP)_s)_{s \in sorts(Sig(SP))}$ is an S-sorted set with one set of values for each sort.

Because test set selection is out of the scope of this thesis, test sets are presented here mostly for exemplifying specifications with testing interface. They are not intended to be the most effective ones.

***Example* 4.60 (Testing Interface)** Here FUNCTION, EXPRESSION, SUBSTITUTION and UNIFICATION, given in Example 3.30, are extended to specifications with testing interface by using a CASL-like notation with an additional construction, namely **test set** to allow test sets to be specified. The testing interface has to supply one test set for each sort in the exported signature of the specification. In other to avoid unjustified repetition, axioms are omitted in the sequel. We use the language of set theory to present test sets. In a real specification language this would have to be given a precise semantics.

spec FUNCTION =
  NAT
and
  STRING
then

sort *function*;

ops $declare\ :\ String \times Nat \rightarrow function;$
$\quad arity\ :\ function \rightarrow Nat;$

vars $n\ :\ String;$
$\quad i\ :\ Nat$

- $arity(declare(n,i)) \ = \ i$

test set

$TF_{Nat} = \{0,3,5\};$

$TF_{String} = \{\text{"f1"},\text{"f2"}\};$

$TF_{function} = \emptyset$

end

Due to the *then* operation, the axiom introduced in FUNCTION is checked with $TF_{\{Nat,String\}}$ by structured testing and with the resulting test set $Test(\text{FUNCTION})_{\{Nat,String\}}$ by flat testing. According to Definition 4.57, $Test(\text{FUNCTION})$ is as follows:

- $Test(\text{FUNCTION})_{Nat} = Test(\text{NAT})_{Nat} \cup TF_{Nat}$ and
- $Test(\text{FUNCTION})_{String} = Test(\text{STRING})_{String} \cup TF_{String}$ and
- $Test(\text{FUNCTION})_{function} = TF_{function}$

Because no *function* values are required to check the FUNCTION axiom, the corresponding test set $TF_{function}$ is defined as empty. In the sequel, empty test sets will be omitted from the test interface for the sake of presentation. The EXPRESSION specification is extended as follows.

spec EXPRESSION =

FUNCTION **and**

CONST **and**

LIST [EXP **fit sort** Elem $\mapsto$ expression] **and**

FINITESET[VAR **fit sort** Elem $\mapsto$ variable]

then

ops $mk\_c\ :\ constant \rightarrow expression;$
$\quad mk\_v\ :\ variable \rightarrow expression;$
$\quad mk\_f\ :\ function \times List[expression] \rightarrow expression;$
$\quad variables\ :\ expression \rightarrow FinSet[variable];$
$\quad variablesl\ :\ List[expression] \rightarrow FinSet[variable];$

preds is_valid : $expression$;
is_validl : $List[expression]$;
vars
...
test set

$TE_{function} = \{declare(\text{"}f\text{"},0), declare(\text{"}g\text{"},1), declare(\text{"}h\text{"},0), declare(\text{"}k\text{"},1), declare(\text{"}v\text{"},2), declare(\text{"}w\text{"},2)\}$;

$TE_{constant} = \{'a', 'b', 'c'\}$

$TE_{variable} = \{'x', 'y', 'z'\}$

$TE_{expression} =$
$\{mk\_f(declare(\text{"}f\text{"},0), []),$
$mk\_f(declare(\text{"}g\text{"},1), [mk\_c('a')]),$
$mk\_f(declare(\text{"}g\text{"},1), [mk\_f(declare(\text{"}h\text{"},0), [])]),$
$mk\_f(declare(\text{"}k\text{"},1), [mk\_v('x')]),$
$mk\_f(declare(\text{"}v\text{"},2), [mk\_v('x'), mk\_v('y')]),$
$mk\_f(declare(\text{"}w\text{"},2), [mk\_f(declare(\text{"}g\text{"},1), [mk\_v('y')]), mk\_c('a')])\}$

$TE_{List[expression]} = \{[e_1 \dots e_n] \mid e_1 \dots e_n \in TE_{expression},\ n \leq 5\}$;

hide
$variablesl, is\_validl$
end

where TE_{Nat}, TE_{String} and $TE_{FiniteSet[variable]}$ are empty because no values of the corresponding sorts are needed to check the axioms introduced in EXPRESSION. Note that the purpose of having specific test sets defined in the testing interface of EXPRESSION for sorts introduced in its subspecifications is to make it possible for specific test data to be considered, possibly of particular interest in EXPRESSION only. Test sets are usually selected according to the axioms (test cases) to be checked. For example, it may be interesting to consider both valid and invalid expressions due to the *is_valid* function. Also, $TF_{function}$ is empty.

The SUBSTITUTION specification is extended as follows.

spec SUBSTITUTION =
EXPRESSION
then
sort $substitution$;

ops *empty* : *substitution*;
dom : $substitution \rightarrow FinSet[variable]$;
rng : $substitution \rightarrow FinSet[variable]$;
add_s : $variable \times expression \times substitution \rightarrow substitution$;
apply : $expression \times substitution \rightarrow expression$;
applyl : $List[expression] \times substitution \rightarrow List[expression]$;
compose : $substitution \times substitution \rightarrow substitution$;
vars
...
test set
$TS_{function} = \{declare(\text{"}f\text{"},0), declare(\text{"}g\text{"},1), declare(\text{"}h\text{"},2), declare(\text{"}f\text{"},3)\}$
$TS_{constant} = \{'a','b'\}$
$TS_{variable} = \{'x','y','z','i','j','k','v','w'\}$
$TS_{expression} = tc \cup tv \cup \bigcup_{i=0...3} tf_i, where:$
$tc = \{mk\_c(c) | c \in TS_{constant}\}$
$tv = \{mk\_v(v) | v \in TS_{variable}\}$
$tf_0 = \{mk\_f(f,[e_1 \ldots e_{arity(f)}]) \mid f \in TS_{function} \text{ and } e_1 \ldots e_{arity(f)} \in tc \cup tv\}$
...
$tf_k = \{mk\_f(f,[e_1 \ldots e_{arity(f)}]) \mid f \in TS_{function} \text{ and } e_1 \ldots e_{arity(f)} \in tf_{k-1}\}$
$TS_{List[expression]} = \{[e_1 \ldots e_n] \mid e_1 \ldots e_n \in TS_{expression},\ n \leq 3\};$
$TS_{substitution} = \bigcup_{i=0...4} ts_i, where:$
$ts_0 = \{empty\}$
$ts_k = \{add\_s(v,e,s) \mid v \in TS_{variable} \wedge e \in TS_{expression} \wedge s \in ts_{k-1}\}$
hide
applyl
end

where lots of different valid expressions and variables seem to be necessary in order to exercise the SUBSTITUTION axioms and define a reasonable number of different substitutions.

spec UNIFICATION =
SUBSTITUTION
then
op *unify* : $expression \times expression \rightarrow substitution$;
preds *is_moregeneral* : $substitution \times substitution$;
is_idempotent : *substitution*;

axiom

...

test set

$TU_{expression} = \{e \in TS_{expression} \mid$
$\exists s \in TS_{substitution}; e' \in TS_{expression} \cdot apply(e,s) = apply(e',s)\}$

$TU_{substitution} = TS_{substitution}$

hide

is_moregeneral, *is_idempotent*

end

where expressions are restricted to the ones which can be unified with another according to a given substitution in the test set of *substitution*. □

4.7.2 Structured Testing

Structured testing consists of a number of tests recursively formulated according to the semantics of structured specifications. In other words, structured testing verifies whether a given program is a checkable model of a specification for a given pair of families of equalities. Specific test sets and equalities are applied to axioms depending on their position in the structure. Also, hidden axioms are checked which presumes that an implementation of hidden symbols is provided together with the program under test.

***Example* 4.61 (Unification *continued*)** Consider again Examples 3.30 and 4.60 and the families of equalities $\approx$ and $\simeq$ in Subsection 4.6.1. Whenever there is no chance of confusion, specification names are also used to refer to their signatures. Suppose we want to see if a given program A is a checkable model of UNIFICATION. According to Definition 4.28:

$$\begin{aligned} &A \in ChMod_{\approx,\simeq}(\text{UNIFICATION})\ \textit{if and only if} \\ &A + A_U \in ChMod_{\approx,\simeq}(\text{UNIFICATION-H}) \end{aligned} \tag{4.1}$$

where UNIFICATION-H is the UNIFICATION specification without hiding the operations *is_moregeneral* and *is_idempotent*, and A_U is an implementation of these operations. Then, by Definition 4.57:

$$A + A_U \in ChMod_{\approx,\simeq}(\textsc{Unification-H}) \; if \; and \; only \; if \\ (A + A_U)|_{\textsc{Substitution}} \in ChMod_{\approx,\simeq}(\textsc{Substitution}) \; and \\ A + A_U \models^{TU}_{\simeq,\simeq} \Phi_U \tag{4.2}$$

where Φ_U is the set of axioms introduced in the UNIFICATION-H specification. Then, by Definition 4.28:

$$(A + A_U)|_{\textsc{Substitution}} \in ChMod_{\approx,\simeq}(\textsc{Substitution}) \\ if \; and \; only \; if \\ (A + A_U)|_{\textsc{Substitution}} + A_S \in ChMod_{\approx,\simeq}(\textsc{Substitution-H}) \tag{4.3}$$

where SUBSTITUTION-H is the SUBSTITUTION specification without hiding the operation *applyl* and A_S is an implementation of this operation. Then, by Definition 4.57:

$$(A + A_U)|_{\textsc{Substitution}} + A_S \in ChMod_{\approx,\simeq}(\textsc{Substitution-H}) \\ if \; and \; only \; if \\ ((A + A_U)|_{\textsc{Substitution}} + A_S)|_{\textsc{Expression}} \in \\ ChMod_{\approx,\simeq}(\textsc{Expression}) \; and \\ (A + A_U)|_{\textsc{Substitution}} + A_S \models^{TS}_{\simeq,\simeq} \Phi_S \tag{4.4}$$

where Φ_S is the set of axioms introduced in the SUBSTITUTION-H specification. Then, by Definition 4.28:

$$((A + A_U)|_{\textsc{Substitution}} + A_S)|_{\textsc{Expression}} \in \\ ChMod_{\approx,\simeq}(\textsc{Expression}) \\ if \; and \; only \; if \\ ((A + A_U)|_{\textsc{Substitution}} + A_S)|_{\textsc{Expression}} + A_E \in \\ ChMod_{\approx,\simeq}(\textsc{Expression-H}) \tag{4.5}$$

where EXPRESSION-H is the EXPRESSION specification without hiding the operations *variablesl* and *is_validl*, and A_E is an implementation of these operations. Then, by Definition 4.57:

$$
\begin{aligned}
&((A+A_U)|_{\text{SUBSTITUTION}} + A_S)|_{\text{EXPRESSION}} + A_E \in \\
&\qquad\qquad\qquad ChMod_{\eqsim,\simeq}(\text{EXPRESSION-H}) \\
&\quad \mathit{if\ and\ only\ if} \\
&(((A+A_U)|_{\text{SUBSTITUTION}} + A_S)|_{\text{EXPRESSION}} + A_E)|_{\text{BAND}} \in \\
&\qquad\qquad\qquad ChMod_{\eqsim,\simeq}(\text{BAND}) \\
&\qquad \mathit{and} \\
&((A+A_U)|_{\text{SUBSTITUTION}} + A_S)|_{\text{EXPRESSION}} + A_E \models^{TE}_{\eqsim,\simeq} \Phi_E
\end{aligned}
\tag{4.6}
$$

where Φ_E is the set of axioms introduced in the EXPRESSION specification and BAND corresponds to:

FUNCTION **and** CONST **and**
LIST[EXP **fit sort** $Elem \mapsto expression$] **and**
FINITESET[VAR **fit sort** $Elem \mapsto variable$]

Let B represent $(((A+A_U)|_{\text{SUBSTITUTION}} + A_S)|_{\text{EXPRESSION}} + A_E)|_{\text{BAND}}$. Then, by Definition 4.58:

$$
\begin{aligned}
&B \in ChMod_{\eqsim,\simeq}(\text{BAND}) \ \mathit{if\ and\ only\ if} \\
&\quad B|_{\text{FUNCTION}} \in ChMod_{\eqsim,\simeq}(\text{FUNCTION}) \ \mathit{and} \\
&\quad B|_{\text{CONST}} \in ChMod_{\eqsim,\simeq}(\text{CONST}) \ \mathit{and} \\
&\quad B|_{\text{LIST[EXP]}} \in ChMod_{\eqsim,\simeq}(\text{LIST[EXP]}) \ \mathit{and} \\
&\quad B|_{\text{FINITESET[VAR]}} \in ChMod_{\eqsim,\simeq}(\text{FINITESET[VAR]})
\end{aligned}
\tag{4.7}
$$

Then by Definition 4.57:

$$
\begin{aligned}
&B|_{\text{FUNCTION}} \in ChMod_{\eqsim,\simeq}(\text{FUNCTION}) \ \mathit{if\ and\ only\ if} \\
&(B|_{\text{FUNCTION}})|_{\text{NAT}} \in ChMod_{\eqsim,\simeq}(\text{NAT}) \ \mathit{and}\ B|_{\text{FUNCTION}} \models^{TF}_{\eqsim,\simeq} \Phi_F
\end{aligned}
\tag{4.8}
$$

where Φ_F is the set of axioms introduced in the FUNCTION specification. Likewise, by Definition 4.13:

$$
B|_{\text{CONST}} \in ChMod_{\eqsim,\simeq}(\text{CONST}) \ \mathit{if\ and\ only\ if}\ B|_{\text{CONST}} \models^{TC}_{\eqsim,\simeq} \Phi_C \tag{4.9}
$$

where Φ_C is the set of axioms introduced in the CONST specification. In this case, Φ_C is empty. Thus, the condition above is always true. Notice that, by the definition of generic specifications and fitting parameters in CASL, LIST[EXP] is a shortened form of:

$$[\text{ELEM } \textbf{with } \sigma_{\text{EXP}}] \textbf{ then } \text{ELEM } \textbf{then } \text{LIST}[\text{ELEM}]$$

where σ_{EXP} is the fitting mapping from *Elem* to *expression*. Likewise FINITESET[VAR] is a shortened form of:

$$[\text{ELEM } \textbf{with } \sigma_{\text{VAR}}] \textbf{ then } \text{ELEM } \textbf{then } \text{FINITESET}[\text{ELEM}]$$

where σ_{VAR} is the fitting mapping from *Elem* to *variable*. For the sake of simplicity, it is assumed that the following holds:

$$B|_{\text{LIST[EXP]}} \in ChMod_{\approx,\simeq}(\text{LIST[EXP]}) \textit{ and}$$
$$B|_{\text{FINITESET[VAR]}} \in ChMod_{\approx,\simeq}(\text{FINITESET[VAR]}) \textit{ and}$$
$$(B|_{\text{FUNCTION}})|_{\text{NAT}} \in ChMod_{\approx,\simeq}(\text{NAT})$$

Then by (4.2), (4.4), (4.6) and (4.8), structured testing corresponds to the following tests:

$$A \in ChMod_{\approx,\simeq}(\text{UNIFICATION}) \textit{ if and only if}$$
$$A + A_U \models^{TU}_{\simeq,\simeq} \Phi_U \textit{ and}$$
$$(A + A_U)|_{\text{SUBSTITUTION}} + A_S \models^{TS}_{\simeq,\simeq} \Phi_S \textit{ and}$$
$$((A + A_U)|_{\text{SUBSTITUTION}} + A_S)|_{\text{EXPRESSION}} + A_E \models^{TE}_{\approx,\simeq} \Phi_E \textit{ and}$$
$$((A + A_U)|_{\text{SUBSTITUTION}} + A_S)|_{\text{EXPRESSION}} + A_E)|_{\text{FUNCTION}} \models^{TF}_{\approx,\simeq} \Phi_F$$

As mentioned in Example 3.30, the *is_moregeneral* axiom has one positive occurrence of $\forall$ and one positive and one negative occurrence of $\exists$. So, we assume that Φ_U contains only the direction of implication of this axiom where the occurrence of $\exists$ is negative so that Corollary 4.55 can be applied. Otherwise, *SP* has to be unbiased for $A, \sim, \doteq$ (see Theorem 4.47). In both cases only incorrectness can be detected when the above tests are performed, unless either UNIFICATION is valid and unbiased or all test sets are exhaustive.

In order to perform these tests it is necessary to supply A_U, A_S and A_E – an implementation of the hidden operations – together with the program under test. Obviously, *variablesl*, *is_validl* and *applyl* are easy to implement. However, *is_moregeneral* and *is_idempotent* may demand much more effort. In addition, a complete and a sound family of equalities with one equality for each test is needed, that is, a pair of approximate equalities for each of the following signatures and their sorts: UNIFICATION-H, SUBSTITUTION-H, EXPRESSION-H and FUNCTION. Suppose these correspond to the families defined in Subsection 4.6.1. An oracle in SML for the first axiom in the first test can be defined as follows.

```
fun unify_o = o_forall TU_expression (fn e =>
 o_forall TU_expression (fn e' =>
  o_exists TU_substitution (fn s =>
    eqs_U(apply(e,s),apply(e',s))) implies
    ((eqc_U(apply(e,unify(e,e')),apply(e',unify(e,e'))))
          andalso
     (o_forall TU_substitution (fn s' =>
             eqs_U(apply(e,s'),apply(e',s')) implies
             is_moregeneral(unify(e,e'),s')))
          andalso
     is_idempotent(unify(e,e')) ) ))
```

where `TU_expression` and `TU_substitution` are the test sets defined in the interface of the UNIFICATION-H specification, `eqc_U` is a complete family of equalities on UNIFICATION-H, with one equality for each sort, and `eqs_U` is a sound one. In addition, note that hidden crucial contexts like *is_moregeneral*(z) and *is_idempotent*(z) may be used to define the complete equality on *substitution* to be applied in the first test. □

4.7.3 Flat Testing

Flat testing consists in checking whether the unstructured set of visible axioms of a structured specification holds in an algebra when considering the resulting test set and two given families of equalities. Hidden axioms cannot be tested and, in order to interpret the results of tests, families of equalities are required to be compatible. Test sets must also be consistent throughout the structure of the specification, in the sense that they may need to coincide for the same sort in different signatures at each occurrence of *union* and, consequently, at each occurrence of *then* and *and*. Also, the specification may be required to be transparent or visible w.r.t. test sets.

***Example* 4.62 (Unification *continued*)** Look again at Examples 3.30 and 4.60 and consider flat testing instead of structured testing in Example 4.61. A program A satisfies UNIFICATION by flat testing requires:

$$A \models^{Test(\text{UNIFICATION})}_{\approx,\simeq} TAx(\text{UNIFICATION}) \tag{4.10}$$

where, by Definitions 4.28 and 4.59:

$$\begin{aligned} TAx(\text{UNIFICATION}) &= TAx(\text{UNIFICATION-H}) \cap Sen(\text{UNIFICATION}) \\ Test(\text{UNIFICATION}) &= Test(\text{UNIFICATION-H}) \cap T_{\text{UNIFICATION}} \end{aligned} \tag{4.11}$$

where UNIFICATION-H is the UNIFICATION specification without hiding the operations *is_moregeneral* and *is_idempotent*. Notice that (4.11) establishes that axioms which refer to these hidden operations are not included in $TAx(\text{UNIFICATION})$. Similarly for test sets. Then, by Definitions 4.57 and 4.59:

$$\begin{aligned} TAx(\text{UNIFICATION-H}) &= TAx(\text{SUBSTITUTION}) \cup \Phi_U \\ Test(\text{UNIFICATION-H}) &= Test(\text{SUBSTITUTION}) \cup TU \end{aligned} \tag{4.12}$$

where Φ_U is the set of axioms introduced in the UNIFICATION-H specification. Then, by Definitions 4.28 and 4.59:

$$\begin{aligned} TAx(\text{SUBSTITUTION}) &= TAx(\text{SUBSTITUTION-H}) \cap \\ &\qquad Sen(\text{SUBSTITUTION}) \\ Test(\text{SUBSTITUTION}) &= Test(\text{SUBSTITUTION-H}) \cap T_{\text{SUBSTITUTION}} \end{aligned} \tag{4.13}$$

where SUBSTITUTION-H is the SUBSTITUTION specification without hiding the operation *applyl*. Then, by Definitions 4.57 and 4.59:

$$\begin{aligned} TAx(\text{SUBSTITUTION-H}) &= TAx(\text{EXPRESSION}) \cup \Phi_S \\ Test(\text{SUBSTITUTION-H}) &= Test(\text{EXPRESSION}) \cup TS \end{aligned} \tag{4.14}$$

where Φ_S is the set of axioms introduced in the SUBSTITUTION-H specification. Then, by Definitions 4.28 and 4.59:

$$\begin{aligned} TAx(\text{EXPRESSION}) &= TAx(\text{EXPRESSION-H}) \cap Sen(\text{EXPRESSION}) \\ Test(\text{EXPRESSION}) &= Test(\text{EXPRESSION-H}) \cap T_{\text{EXPRESSION}} \end{aligned} \tag{4.15}$$

where EXPRESSION-H is the EXPRESSION specification without hiding the operation *applyl*. Finally, by Definitions 4.57, 4.58 and 4.59 and obvious simplifications:

$$\begin{aligned} TAx(\textsc{Expression-H}) = \;& TAx(\textsc{Function}) \cup \Phi_E \\ Test(\textsc{Expression-H}) = \;& Test(\textsc{Function}) \cup \\ & Test(\textsc{Const}) \cup \\ & Test(\textsc{List}[\textsc{Exp}]) \cup \\ & Test(\textsc{FiniteSet}[\textsc{Var}]) \cup \\ & TE \end{aligned} \tag{4.16}$$

$$Test(\textsc{Function}) = Test(\textsc{Nat}) \cup Test(\textsc{String}) \cup TF \tag{4.17}$$

where Φ_E is the set of axioms introduced in the EXPRESSION-H specification. In (4.16), we assume that axioms in LIST[EXP] and FINITESET[VAR] always hold, so they are not considered.

It can be noticed that $TAx(\textsc{Unification})$ does not include any axiom referring to the *function* sort whose operations are defined in terms of hidden axioms. Likewise, the main unification axiom cannot be checked because it refers to hidden operations, which is an extreme drawback of flat testing when used for the UNIFICATION specification.

Now consider Corollary 4.56 and the sound family of equalities $\simeq$ defined in Subsection 4.6.1. Notice that UNIFICATION does not preserve encapsulation, since *is_moregeneral* and *is_idempotent* are *substitution* observers added in UNIFICATION-H by the *then* operation. However, a complete and compatible family of equalities, namely $\overline{\approx}$, can be defined, in this particular case and focusing on the signatures arising in UNIFICATION, by not taking the observers which cause the problem as crucial contexts. Nevertheless, it is important to remember that the compatibility conditions on families of equalities are due to *translate* and *hide*. Suppose *is_moregeneral* and *is_idempotent* are introduced in SUBSTITUTION-H. Then UNIFICATION does not preserves encapsulation again since these symbols are hidden. Therefore, *hide* together with *and*, *then* and *translate* should always be carefully taken into account.

In addition, in order to interpret the results of flat testing, it is also necessary to check whether the UNIFICATION specification is consistent w.r.t. test sets. Thus, the following must coincide:

1. $Test(\textsc{Nat})_{Nat}$ and TF_{Nat} from (4.17)

2. $Test(\textsc{String})_{String}$ and TF_{String} from (4.17)

3. $Test(\textsc{Function})_{\{Nat,String\}}$ and $TE_{\{Nat,String\}}$ from (4.16)

4. $Test(\textsc{Const})_{constant}$ and $TE_{constant}$ from (4.16)

5. $Test(\textsc{FiniteSet}[\textsc{Var}])_{variable}$ and $TE_{variable}$ from (4.16)

6. $Test(\textsc{Expression})_{\{Nat,String,constant,variable,expression\}}$ and $TS_{\{Nat,String,constant,variable,expression\}}$ from (4.14)

7. $Test(\textsc{Substitution})_{\{Nat,String,constant,variable,expression,substitution\}}$ and $TU_{\{Nat,String,constant,variable,expression,substitution\}}$ from (4.12)

Notice that some simplifications were applied above by relying only upon sorts referred to by testable axioms. For instance, $Test(\textsc{Function})_{function}$ and $TE_{function}$ need not coincide, since *function* is only referred to by hidden axioms in the EXPRESSION-H specification. The same applies to $Test(\textsc{Expression})_{function}$ and $TS_{function}$. Also, due to Definition 4.34, $TS_{substitution}$ need not be considered w.r.t. EXPRESSION.

Clearly the UNIFICATION specification as presented in Example 4.60 may not be consistent w.r.t. test sets. The conditions above may not hold, except 6 which holds partially, as $TU_{substitution} \stackrel{\text{def}}{=} TS_{substitution} = Test(\textsc{Substitution})_{substitution}$. Thus, in order to perform flat testing, it is necessary to modify the specification, so that TF, TE, TS and TU are defined according to the rules above. This illustrates another limitation of flat testing regarding lack of flexibility to define test sets. In fact, flat testing can make testing interfaces look like a bit awkward and not very interesting. For example, $TE_{\{Nat,String,constant,variable\}}$ must be defined as follows:

$$TE_{Nat} = Test(\textsc{Function})_{Nat};$$
$$TE_{String} = Test(\textsc{Function})_{String};$$
$$TE_{constant} = Test(\textsc{Const})_{constant};$$
$$TE_{variable} = Test(\textsc{FiniteSet}[\textsc{Var}])_{variable};$$

where only $TE_{function}$, $TE_{expression}$, $TE_{List[expression]}$ and $TE_{FiniteSet[variable]}$ can be chosen independently of their counterparts in FUNCTION, LIST[EXPRESSION] and FINITESET[VAR], insofar as these sorts are not referred to by any visible axiom of EXPRESSION. On the other hand, $TS_{expression}$ must coincide with its counterpart $Test(\textsc{Expression})_{expression}$ which is not necessarily equal to $TE_{expression}$, insofar as $Test(\textsc{List}[\textsc{Exp}])_{expression}$ may not be empty and it need not coincide with $TE_{expression}$.

Finally, an oracle for checking the UNIFICATION specification by flat testing can be defined in SML as follows:

```
fun unification_o =
  function_o andalso expression_o andalso substitution_o

val function_o =
   o_forall T_string (fn n => o_forall T_int (fn i =>
                              arity(declare(n,i)) = i))
val expression_o =
  o_forall T_constant (fn c =>
    (is_valid(mk_c(c))) andalso
    (variables(mk_c(c)) = emptyset)) andalso
  o_forall T_variable (fn v =>
    (is_valid(mk_v(v))) andalso
    (variables(mk_v(v)) = add(v,emptyset)))

val substitution_o =
  (o_forall T_variable (fn x =>
   o_forall T_expression (fn e =>
   o_forall T_substitution (fn s =>
     (not (elemOf(x,dom(s))) andalso
      not (eqc_U(e,mk_v(x)))) implies
        (dom(add_s(x,e,s)) =
           (union(add(x,emptyset),dom(s))) andalso
         rng(add_s(x,e,s)) =
           (union(variables(e),rng(s)))))))))) andalso
  (o_forall T_expression (fn e => apply(e,empty) = e))
  ...
  (o_forall T_substitution (fn s =>
   o_forall T_substitution (fn s' =>
   o_forall T_substitution (fn s'' =>
     (eqc_U(compose(e,empty),s) andalso
        eqc_U(compose(empty,s),s)) andalso
     (eqc_U(compose(compose(s,s'),s''),
            compose(s,compose(s',s'')))))))))
```

where `eqc_U` is a complete family of equalities on sorts of UNIFICATION and `T_s` is $\mathit{Test}(\text{UNIFICATION})_s$ for $s \in \mathit{sorts}(\text{UNIFICATION})$. For the sake of simplicity, it is assumed that $\mathit{FiniteSet}[\mathit{variable}]$ is an observable sort. □

4.8 Concluding Remarks

In order to overcome the oracle problem in the context of structured specifications, restrictions on both specifications and test suites may be needed. For flat specifications, the use of sound/complete approximate equalities and assumptions on the presence/absence of quantifiers are required. However, structured specifications introduce further complications to the oracle problem. For instance, equalities over different signatures might be needed for tackling different parts of the specification and these might be required to be compatible with signature morphisms. Also, different test sets for different parts of a structured specification can be defined. Furthermore, hidden symbols may be defined in the specification and not necessarily implemented in the program under test.

Contrary to flat testing, provided that families of approximate equalities which are sound/complete can be defined, previous results on the use of approximate equalities presented in Chapter 3 can be generalised for structured testing without any additional restrictions since structured testing is based on the compositional semantics of specifications. Structured testing demands an implementation of hidden symbols which is a reasonable requirement, but it cannot always be met.

Flat testing is more limited than structured testing (see Figure 4.1). For instance, in order to interpret test results, specifications are required to be consistent w.r.t. test sets which may impede certain tests from being performed. Moreover, these specifications may need to be transparent, that is, have no occurrences of *hide*, unless either hidden axioms can be replaced by their visible consequences or the specification can be normalised, which may be complicated for large specifications, not to mention the loss of structure. Furthermore, flat testing can only be interpreted if the equalities considered belong to families which are compatible w.r.t. signature morphisms. In the observational case, signatures may not be allowed to add observers for sorts previously introduced in other signatures.

Chapter 5

The Rôle of Normalization

"More than the act of testing, the act of designing tests is one of the most effective bug preventers known."
B. Beizer

Contents

This chapter investigates the use of normalisation for testing from structured algebraic specifications. The intention is to compute a related, possibly simpler, specification, namely the *normal form*, and use it to test programs as an alternative to approaches presented earlier. In order to address the oracle problem, two extreme styles of testing are presented in Chapter 4, namely *structured testing* and *flat testing*. Whenever neither structured nor flat testing can be effectively applied, a combined style based on the use of normal forms, namely *semi-structured testing*, can be considered instead. In this chapter, three normal forms are presented for coping with some obstacles encountered when testing from structured specifications [Machado, 2000a]. Furthermore, it turns out that, under certain circumstances, tests based on normal forms can be more rigorous and effective than tests based on the original specification.

5.1 Normalisation and Testing

Normalisation of ordinary specifications as presented in [Bidoit et al., 1999], based on the laws of module algebra [Bergstra et al., 1990], aims at producing a flat version of a structured specification, the normal form, which is equivalent to the original w.r.t signature and model class. There, the intention is to use normalisation to define non-compositional proof systems for deriving theorems of a specification from a flat set of its axioms by using some standard proof system of the underlying institution. On the other hand, compositional proof systems perform derivations according to the modular structure of specifications, allowing proofs to be constructed in a structured way. The drawback of normalisation is the loss of structure which can be crucial for large specifications. Nevertheless, in practice, one may need to combine the two approaches so that normal forms are computed for some parts of the specification and compositional proof systems are considered for the overall specification.

According to Chapter 4, testing from structured specifications can be done either in a pure compositional or non-compositional way, namely *structured testing* and *flat testing* respectively, where structured testing is more flexible than flat testing in the sense that oracles can be given under fewer assumptions on tests. Structured testing corresponds to membership in the class of checkable models, whereas flat testing corresponds to testing satisfaction of visible axioms according to the resulting test set computed from the specification. Structured testing may require different parts of the specification to be checked separately and also an implementation of hidden sorts and functions. Again, structured and flat testing are two extreme ways of testing structured specifications and, in practice, many advantages can arise from combining them. Normalisation appears to be a way of making it possible to perform "semi-structured" testing, where programs are tested against specifications in a compositional way, but some parts of specifications are replaced by their corresponding normal forms. Also, normalisation gives rise to a non-compositional style of testing in which oracles can be given under fewer assumptions, and so, dealing more effectively with the oracle problem. Moreover, hidden axioms can be more appropriately handled as hidden definitions are grouped. Furthermore, under certain circumstances, normalisation can make testing experiments more rigorous in the sense of reducing the number of incorrect programs accepted and correct programs rejected.

In the sequel, three normal forms are presented. Testing from the original specification, pure structured testing, and flat testing are compared to testing from the normal forms, showing under which circumstances the conclusions of Theorems

4.45 and 4.46 and Theorems 4.51 and 4.52 can be met when taking the normal forms into account. Example 4.22 is used to illustrate the normal forms. Moreover, the basis for effectively testing from specifications composed of structured specifications and normal forms is provided. As in Chapter 4 (see Definition 4.18), structured specifications are defined using the specification-building operations *union*, *translate* and *hide* [Wirsing, 1990, Sannella and Tarlecki, 1997, Hennicker, 1997].

This chapter is structured as follows. Section 5.2 presents the ***con*** normal form which increments test sets in the testing interface at each occurrence of the union operation. Section 5.3 describes an extended version of the normal form presented in [Bidoit et al., 1999], namely the ***nf*** normal form, for specifications with testing interface. Section 5.4 presents the strict normal form (***snf***) where each axiom is explicitly associated with a test set.

5.2 The *con* Normal Form

When testing from a specification *SP* by structured testing, axioms are checked according to the test set defined in the subspecification they belong to. On the other hand, flat testing considers only $Test(SP)$, the visible part of the combined test set. Clearly, axioms can be checked with different test sets in the former and in the second approach, mainly when *SP* has occurrences of *union* (see Example 4.44). Unless specifications are consistent w.r.t test sets, due to *union* and *translate*, and *hide* is taken into account, an oracle in the second approach cannot be given (see Theorems 4.51 and 4.52).

If *SP* is compatible w.r.t test sets, the ***con*** normal form produces a new specification from *SP* which is consistent w.r.t. test sets by incrementing test sets of each sort at each occurrence of the *union* operation, while keeping the structure of the former specification. The intention is to allow non-compositional approaches like flat testing to be performed from the ***con*** normal form so that they can be appropriately interpreted according to the class of real models. Nevertheless, structured testing can also take advantage of incrementing test sets, since specifications with testing interface can be incremented by additional test sets without compromising the final interpretation, provided that assumptions on quantifiers are made (see Theorem 3.20). Actually, adding test sets can make the testing experiment more rigorous. This might seem to be obvious if the specification axioms have only positive $\forall$ (and negative $\exists$). However, this is less obvious when the specification has only positive $\exists$ (and negative $\forall$), since, in this context, more programs are accepted when test sets are incremented (see Theorem 3.20). Specifications can be incremented by test sets as follows.

Definition 5.1 (Incremented Specifications) *Let SP be a structured specification and T' be a $Sig(SP)$-test set.*

1. *If* $SP = \langle \Sigma, \Phi, T \rangle$, *then* $\boldsymbol{inc}(SP, T') = \langle \Sigma, \Phi, T \cup T' \rangle$
2. *If* $SP = SP_1 \cup SP_2$, *then* $\boldsymbol{inc}(SP, T') = \boldsymbol{inc}(SP_1, T') \cup \boldsymbol{inc}(SP_2, T')$
3. *If* $SP =$ translate SP' by σ, *then* $\boldsymbol{inc}(SP, T') =$ translate $\boldsymbol{inc}(SP', \{t \in T_{\Sigma'} \mid \sigma(t) \in T'\})$ by σ, *where* $\sigma : \Sigma' \to \Sigma$, $Sig(SP) = \Sigma$ *and* $Sig(SP') = \Sigma'$
4. *If* $SP =$ hide sorts S' opns F' in SP', *then* $\boldsymbol{inc}(SP, T') =$ hide sorts S' opns F' in $\boldsymbol{inc}(SP', T')$

Lemmas 5.2 and 5.3 below relate testing from an incremented specification to structured testing from the original one. Let *SP* be a structured specification, T' be a $Sig(SP)$-test set and A be a $Sig(SP)$-algebra.

Lemma 5.2 *If the axioms of SP have only positive occurrences of* $\forall$ *and negative occurrences of* $\exists$ *then* $A \in ChMod_{\sim,\simeq}(\boldsymbol{inc}(SP, T'))$ *implies* $A \in ChMod_{\sim,\simeq}(SP)$.

Proof. By induction on the structure of *SP* and Definitions 5.1, 4.23, 4.26 and 4.28. The only interesting case to look at is when $SP = \langle \Sigma, \Phi, T \rangle$. Suppose $A \in ChMod_{\sim,\simeq}(\boldsymbol{inc}(SP, T'))$. Then $A \models_{\sim,\simeq}^{T \cup T'} \Phi$. Because Φ only has positive $\forall$ and negative $\exists$, then, by Theorem 3.20, $A \models_{\sim,\simeq}^{T} \Phi$. □

Lemma 5.3 is the dual of Lemma 5.2.

Lemma 5.3 *If the axioms of SP have only negative occurrences of* $\forall$ *and positive occurrences of* $\exists$ *then* $A \in ChMod_{\sim,\simeq}(SP)$ *implies* $A \in ChMod_{\sim,\simeq}(\boldsymbol{inc}(SP, T'))$.

Proof. Follows the same pattern as the proof of Lemma 5.2. □

Obviously, from Theorem 3.20 and under the assumptions stated in Lemmas 5.2 and 5.3, flat testing from incremented specifications can be more effective than from the original one.

The ***con*** normal form given below increments a specification with a test set at each occurrence of the *union* operation so that the resulting specification is consistent w.r.t. test sets if the original one is compatible w.r.t. test sets.

Definition 5.4 (*con* Normal Form) *Let SP be a structured specification. The normal form* $\mathbf{con}(SP)$ *is defined as follows.*

1. $\boldsymbol{con}(\langle\Sigma,\Phi,T\rangle) \stackrel{def}{=} \langle\Sigma,\Phi,T\rangle$

2. $\boldsymbol{con}(SP_1 \cup SP_2) \stackrel{def}{=} \boldsymbol{con}(\boldsymbol{inc}(SP_1,T)) \cup \boldsymbol{con}(\boldsymbol{inc}(SP_2,T))$,
   where $T = Test(SP_1 \cup SP_2)$

3. $\boldsymbol{con}(\textsf{translate } SP' \textsf{ by } \sigma) \stackrel{def}{=} \textsf{translate } \boldsymbol{con}(SP') \textsf{ by } \sigma$

4. $\boldsymbol{con}(\textsf{hide sorts } S' \textsf{ opns } F' \textsf{ in } SP') \stackrel{def}{=} \textsf{hide sorts } S' \textsf{ opns } F' \textsf{ in } \boldsymbol{con}(SP')$

Proposition 5.5 $\boldsymbol{con}(SP)$ *is a well-formed specification.*

The reason why $\boldsymbol{con}(SP)$ is a well-formed specification is that it has the same structure as *SP* and increments in test set occurs only when *union* is reached. At this point, only terms built from visible symbols relative to the corresponding argument specifications and not to the overall specification are added (see item 2 in Definition 5.4, and Definition 5.1). Thus, name clashes cannot occur and test sets in the outer structure can always be defined from test sets in the inner structure. For any signature morphism $\sigma : \Sigma' \to \Sigma$ used with *translate*, sorts $s \in sorts(\Sigma)$ which do not correspond to any sort in $sorts(\Sigma')$ do not have their test sets incremented.

The following are properties of specifications in ***con*** normal form.

Proposition 5.6

1. *If SP is compatible w.r.t test sets, then* $\mathbf{con}(SP)$ *is consistent w.r.t test sets.*

2. *If SP is consistent w.r.t test sets, then* $SP = \mathbf{con}(SP)$.

3. $\boldsymbol{con}(\boldsymbol{con}(SP)) = \boldsymbol{con}(SP)$.

In Example 4.22, $\boldsymbol{con}(IntList)$ is consistent w.r.t. test set if *IntList* is compatible, that is, T_2 is defined from symbols in Σ_1 only. Otherwise, $Test(\boldsymbol{con}(\boldsymbol{inc}(SList1,T)))_{list} \subset Test(\boldsymbol{con}(\boldsymbol{inc}(SList2,T)))_{list}$ with $T = Test(SList1) \cup Test(SList2)$, but $Test(\boldsymbol{con}(\boldsymbol{inc}(SList1,T)))_{list} \neq Test(\boldsymbol{con}(\boldsymbol{inc}(SList2,T)))_{list}$. Also, if $T_1 = T_2$, then *IntList* is consistent w.r.t. test sets.

5.2.1 Structured Testing and the *con* Normal Form

Obviously, the class of real models of a specification *SP* corresponds exactly to the class of real models of $\boldsymbol{con}(SP)$, since $Mod_{\approx}$ does not take test sets into account. However, because *SP* and $\boldsymbol{con}(SP)$ have different test sets, the classes of checkable models of *SP* and $\boldsymbol{con}(SP)$ are not equivalent. The implication in both directions holds only under certain assumptions on quantifiers which are contradictory (see Theorems 5.7 and 5.8 below). Let *SP* be a structured specification and *A* be a $Sig(SP)$-algebra.

Theorem 5.7 *If the axioms of SP have only positive occurrences of $\forall$ and negative occurrences of $\exists$, $A \in ChMod_{\sim,\doteq}(\boldsymbol{con}(SP))$ implies $A \in ChMod_{\sim,\doteq}(SP)$.*

Proof. By induction on the structure of *SP*. The only interesting case to look at is when $SP = SP_1 \cup SP_2$. Suppose $A \in ChMod_{\sim,\doteq}(\boldsymbol{con}(SP))$. Then, by Definitions 5.4 and 4.23, $A \in ChMod_{\sim,\doteq}(\boldsymbol{con}(\boldsymbol{inc}(SP_1,T)))$ and $A \in ChMod_{\sim,\doteq}(\boldsymbol{con}(\boldsymbol{inc}(SP_2,T)))$, where $T = Test(SP_1 \cup SP_2)$. By induction hypothesis, $A \in ChMod_{\sim,\doteq}(\boldsymbol{inc}(SP_1,T))$ and $A \in ChMod_{\sim,\doteq}(\boldsymbol{inc}(SP_2,T))$. Thus, by Lemma 5.2, $A \in ChMod_{\sim,\doteq}(SP_1)$ and $A \in ChMod_{\sim,\doteq}(SP_2)$. Hence, $A \in ChMod_{\sim,\doteq}(SP)$. □

Obviously, the converse holds if the assumptions of Lemma 5.3 are fulfilled.

Theorem 5.8 *If the axioms of SP have only negative occurrences of $\forall$ and positive occurrences of $\exists$, $A \in ChMod_{\sim,\doteq}(SP)$ implies $A \in ChMod_{\sim,\doteq}(\boldsymbol{con}(SP))$.*

Proof. By Definition 5.4 and Lemma 5.3. □

The following corollaries relate the class of checkable models of the normalised specification to the class of real models of the original specification. Notice that if *A* is a real model of *SP*, then, by Corollary 5.9 below (if its conditions hold), *A* is a checkable model of $\boldsymbol{con}(SP)$. By Theorem 5.7, *A* is a checkable model of *SP*. This means that under the assumptions of Corollary 5.9 and Theorem 5.7, it may be more interesting to test from the ***con*** normal form instead of the original specification.

Corollary 5.9 *If $\sim$ is complete, $\doteq$ is sound and the axioms of SP have only positive occurrences of $\forall$ and negative occurrences of $\exists$ then $A \in Mod_{\approx}(SP)$ implies $A \in ChMod_{\sim,\doteq}(\boldsymbol{con}(SP))$.*

Proof. Suppose $A \in Mod_{\approx}(SP)$. Then, $A \in Mod_{\approx}(\boldsymbol{con}(SP))$. Thus, by Theorem 4.45, $A \in ChMod_{\sim,\doteq}(\boldsymbol{con}(SP))$. □

The dual of Corollary 5.9 presented below follows from Theorem 4.46.

Corollary 5.10 *If $\sim$ is sound, $\doteq$ is complete and the axioms of SP have only negative occurrences of $\forall$ and positive occurrences of $\exists$ then $A \in ChMod_{\sim,\doteq}(\textbf{con}(SP))$ implies $A \in Mod_{\approx}(SP)$.*

The normal form $\textbf{\textit{con}}(SP)$ can be more efficient than *SP* in the sense that if the conditions of Theorem 5.7 and Corollary 5.9 are met, then testing from $\textbf{\textit{con}}(SP)$ can accept fewer incorrect programs than testing from *SP* (see Figure 5.1). Similarly, if the conditions of Theorem 5.8 and Corollary 5.10 are met, then testing from $\textbf{\textit{con}}(SP)$ can reject fewer correct programs than testing from *SP*. Nevertheless, the classes of checkable models of *SP* and $\textbf{\textit{con}}(SP)$ are unlikely to be equivalent[1] which implies that $\textbf{\textit{con}}(SP)$ cannot replace *SP* in the structure of another specification, except for the cases covered by Theorem 5.7 and Corollary 5.9 where $Mod_{\approx}(SP) \subseteq ChMod_{\sim,\doteq}(\textbf{con}(SP)) \subseteq ChMod_{\sim,\doteq}(SP)$ and and Theorem 5.8 and Corollary 5.10 where $ChMod_{\sim,\doteq}(SP) \subseteq ChMod_{\sim,\doteq}(\textbf{con}(SP)) \subseteq Mod_{\approx}(SP)$.

$$\begin{tikzcd}
 & A \in Mod_{\approx}(SP) \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] \arrow[d, phantom, "\begin{array}{c} +\forall\ -\exists \\ \sim complete \\ \doteq sound \\ \text{Thm.}(4.45)\ \textit{and}\ \text{Cor.}(5.9) \end{array}"] & \\
A \in ChMod_{\sim,\doteq}(SP) & & A \in ChMod_{\sim,\doteq}(\textbf{con}(SP)) \arrow[ll, Rightarrow, "\begin{array}{c} +\forall\ -\exists \\ \text{Thm.}(5.7) \end{array}"']
\end{tikzcd}$$

Figure 5.1: Testing from the ***con*** normal form and structured testing.

***Example* 5.11** Consider the *IntList* specification given in Example 4.22. From Definition 5.4, the normal form $\textbf{\textit{con}}(IntList)$ is as follows.

$$\begin{array}{l} \mathsf{hide\ sorts}\ \emptyset\ \mathsf{opns}\ \{is\_sorted\}\ \mathsf{in} \\ \quad \mathsf{translate}\ \langle \Sigma_1, \Phi_1, T \cap T_{\Sigma_1} \rangle\ \mathsf{by}\ \sigma \cup \\ \quad \langle \Sigma_2, \Phi_2, T \rangle \end{array}$$

[1] There is no such precise characterisation apart from the apparent case where *SP* is consistent w.r.t test sets, that is, $con(SP) = SP$.

where $T = Test(SList1) \cup Test(SList2) = \sigma(Test(List)) \cup T_2 = T_1 \cup T_2$, the test set $T \cap T_{\Sigma_1}$ is restricted to terms of symbols in Σ_1, Φ_1 is the set of axioms of *List* and Φ_2 is the set of axioms of *SList*2. Let A be an $IntList$-algebra. From Definitions 4.23, 4.26 and 4.28, structured testing now corresponds to the following tests: $A'|_{\Sigma_1} \models^{T \cap T_{\Sigma_1}}_{\sim_{\Sigma_1}, \doteq_{\Sigma_1}} \Phi_1$ and $A' \models^{T}_{\sim_{\Sigma_2}, \doteq_{\Sigma_2}} \Phi_2$ with $A = A'|_{Sig(IntList)}$ (see Example 4.44). Obviously, from Corollary 5.9 and Theorem 5.7, if $\sim$ is complete and $\doteq$ is sound, structured testing from the ***con*** normal form can be more rigorous than from the original specification, since test sets are incremented in the former. □

5.2.2 Flat Testing and the *con* Normal Form

Flat testing can benefit from the ***con*** normal form by weakening the consistency condition on test sets. Let *SP* be a structured specification. It is easy to see that $TAx(SP)$ coincides with $TAx(\mathbf{con}(SP))$, since only test sets are changed, and also $Test(SP)$ coincides with $Test(\mathbf{con}(SP))$, since values which do not belong to the former are not added to the latter.

Lemma 5.12 $Test(\mathbf{con}(SP)) = Test(SP)$.

Proof. By induction on *SP*. The only interesting case to look at is when $SP = SP_1 \cup SP_2$. Let $T = Test(SP_1 \cup SP_2)$.

$Test(\mathbf{con}(SP_1 \cup SP_2))$
$= Test(\mathbf{con}(\mathbf{inc}(SP_1, T)) \cup \mathbf{con}(\mathbf{inc}(SP_2, T)))$, by Definition 5.4
$= Test(\mathbf{con}(\mathbf{inc}(SP_1, T))) \cup Test(\mathbf{con}(\mathbf{inc}(SP_2, T)))$, by Definition 4.23
$= Test(\mathbf{inc}(SP_1, T)) \cup Test(\mathbf{inc}(SP_2, T))$, by induction hypothesis
$= T$, as $Test(SP_1), Test(SP_2) \subseteq T$

□

Accordingly, flat testing from the ***con*** normal form is equivalent to testing from the original specification. However, when testing from the normal form, the consistency condition in Theorems 4.51 and 4.52 can be replaced by compatibility. Thus, the following corollaries are direct consequences of these theorems and Proposition 5.6(1). Let *SP* be a structured specification and A be a $Sig(SP)$-algebra.

Corollary 5.13 *Let SP be compatible w.r.t. test sets. If $\sim$ is complete and compatible and $\doteq$ is sound and compatible and the axioms of SP have only positive occurrences of $\forall$ and negative occurrences of $\exists$, then $A \in Mod_{\approx}(SP)$ implies $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$.*

Proof. Suppose $A \in Mod_{\approx}(SP)$. Then, $A \in Mod_{\approx}(\mathbf{con}(SP))$. Thus, because $\mathbf{con}(SP)$ is consistent w.r.t test sets, by Theorem 4.51, $A \models^{Test(\mathbf{con}(SP))}_{\sim,\doteq} TAx(\mathbf{con}(SP))$. Hence, $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$. □

Corollary 5.14 *Let SP be compatible w.r.t. test sets and transparent. If $\sim$ is sound and compatible and $\doteq$ is complete and compatible and the axioms of SP have only negative occurrences of $\forall$ and positive occurrences of $\exists$, then $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$ implies $A \in Mod_{\approx}(SP)$.*

Therefore, from the above corollaries, the consistency condition in Theorems 4.51 and 4.52 (see Figure 4.1) can be replaced by compatibility (see Figure 5.2). Obviously, flat testing from any two distinct specifications corresponds to the same testing if their resulting test sets and sets of visible axioms are the same. Thus, if *SP* is compatible w.r.t. test sets (and not necessarily consistent) then the conclusions of Theorem 4.51 can be reached. But, in practical terms, this may not represent a big improvement, since the compatible condition itself can also substantially limit the flexibility to define test sets. For instance, the problems mentioned in Subsection 4.5.3 regarding T_2 and the *sort* operation in Example 4.22 still arise (see also Example 4.38, where *reverse* can be exercised if *IntList* is consistent but not compatible).

Example* 5.15** Flat testing from *IntList* (Example 4.44) and from $\mathbf{con}(IntList)$ are exactly the same, since their resulting test sets and sets of visible axioms are the same. However, it comes out from this section that, due to the ***con normal form, *IntList* does not need to be consistent w.r.t. test sets, but only compatible. This means that T_2 is required to be defined in the scope of Σ_1, but it need not be equal to T_1. □

***Example* 5.16 (Unification *continued*)** Consider Example 4.62. In order to interpret the results of flat testing, the UNIFICATION specification is required to be either consistent or compatible w.r.t. test sets. In order to meet the compatibility condition, the following test sets must be defined in the scope of the given signature:

1. TF_{Nat}, TE_{Nat}, TS_{Nat} and TU_{Nat} in the scope of NAT

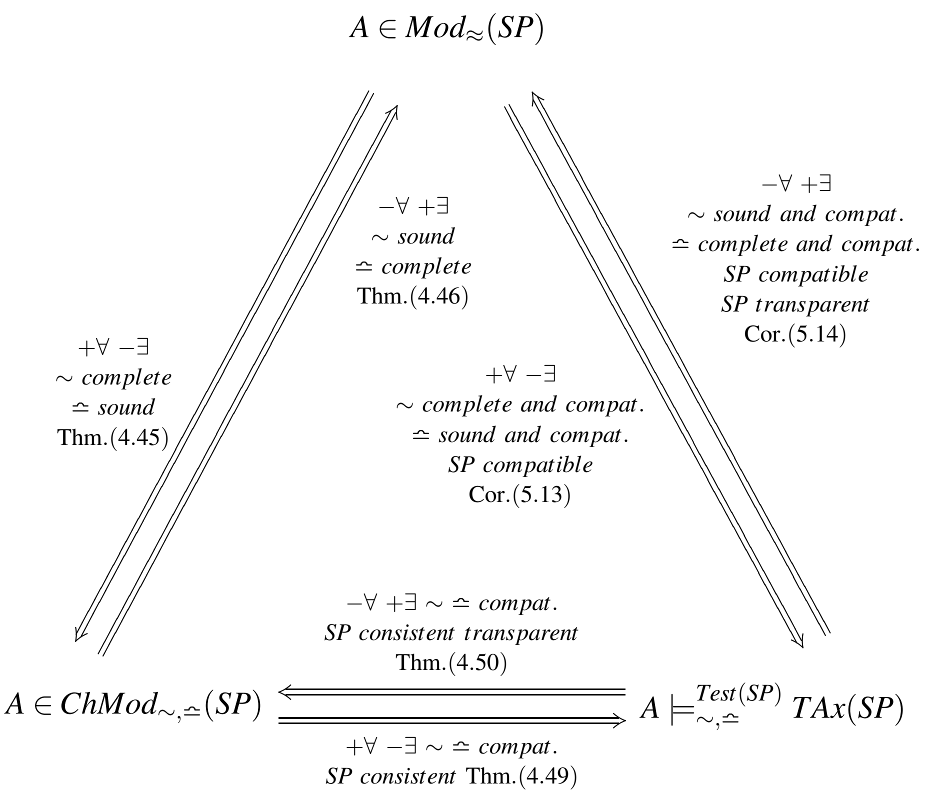

Figure 5.2: Structured testing, flat testing, correctness and the ***con*** normal form.

2. TF_{String}, TE_{String}, TS_{String} and TU_{String} in the scope of STRING

3. $TE_{constant}$, $TS_{constant}$ and $TU_{constant}$ in the scope of CONST

4. $TE_{variable}$, $TS_{variable}$ and $TU_{variable}$ in the scope of FINITESET[VAR]

5. $TS_{expression}$ and $TU_{expression}$ in the scope of LIST[EXP]

6. $TU_{substitution}$ in the scope of SUBSTITUTION

When compared to the conditions stated in Example 4.62, the above mentioned seem to be a lot weaker. At least 1, 2, 3 and 4 are no big deal. However, 5 and 6 can still be quite limiting, since both SUBSTITUTION and UNIFICATION have constructors on *expression* (*apply*) and *substitution* (*unify*) which cannot be exercised otherwise by flat testing since they are defined by hidden axioms. □

5.3 The *nf* Normal Form

The ***nf*** normal form extends the normal form introduced in [Bidoit et al., 1999] for ordinary specifications to deal with structured specifications with testing interface. The intention is to handle the complexity of structured specifications by grouping axioms, taking hidden symbols into account, so that the result is a flat specification which exports visible symbols. The symbols of a specification includes both visible and hidden symbols appropriately renamed to avoid name clashes.

Definition 5.17 (Symbols) *The symbols of a structured specification are defined as follows.*

1. $Symbols(\langle \Sigma, \Phi, T \rangle) \stackrel{def}{=} \Sigma$
2. $Symbols(SP_1 \cup SP_2) \stackrel{def}{=} Symbols(SP_1) +_{Sig(SP)} Symbols(SP_2)$ *(Figure 5.3)*
3. $Symbols(\textsf{translate } SP' \textsf{ by } \sigma) \stackrel{def}{=} PO(Sig(SP') \hookrightarrow Symbols(SP'), \sigma)$, *where* $\sigma : Sig(SP') \rightarrow Sig(SP)$ *(Figure 5.4)*
4. $Symbols(\textsf{hide sorts } S' \textsf{ opns } F' \textsf{ in } SP') \stackrel{def}{=} Symbols(SP')$

where diagrams in Figures 5.3 and 5.4 are pushout constructions chosen to rename all hidden symbols so that $Sig(SP) \subseteq Symbols(SP)$*.*

$$\begin{tikzcd}
Sig(SP) \arrow[r, hook] \arrow[d, hook] & Symbols(SP_2) \arrow[d, dashed, "in_2"] \\
Symbols(SP_1) \arrow[r, dashed, "in_1"] & Symbols(SP_1) +_{Sig(SP)} Symbols(SP_2)
\end{tikzcd}$$

Figure 5.3: $Symbols(SP_1 \cup SP_2)$

In the case of *union*, although the signatures of SP, SP_1 and SP_2 are the same (and this is the reason why the diagram in Figure 5.3 is simpler than the one presented in [Bidoit et al., 1999]), $Symbols(SP_1)$ may not coincide with $Symbols(SP_2)$, because the hidden symbols of SP_1 and SP_2 can be different.

Let $SP = \textsf{hide sorts } S' \textsf{ opns } F' \textsf{ in } SP'$ be a specification. Then $SP = SP'|_\Sigma$ is an abbreviation for this use of *hide*, where $\Sigma = Sig(SP') - (S', F') = Sig(SP)$. This

$$\begin{tikzcd}
Sig(SP') \arrow[r, "\sigma"] \arrow[d, hook] & Sig(SP) \arrow[d, hook, dashed] \\
Symbols(SP') \arrow[r, dashed, "\varsigma"] & PO(Sig(SP') \hookrightarrow Symbols(SP'), \sigma)
\end{tikzcd}$$

Figure 5.4: $Symbols(\textsf{translate } SP' \textsf{ by } \sigma)$

abbreviation is used here rather than *hide* to facilitate comparison with the usual normal form construction.

The ***nf*** normal form of a specification SP is a basic specification having $Symbols(SP)$ as its signature restricted to a signature $Sig(SP)$ of exported symbols defined as follows.

Definition 5.18 (*nf* Normal Form) *Let SP be a structured specification. The normal form* $\mathbf{nf}(SP)$ *is defined as follows.*

1. *If* $SP = \langle \Sigma, \Phi, T \rangle$, *then* $\mathbf{nf}(SP) \stackrel{def}{=} \langle \Sigma, \Phi, T \rangle|_\Sigma$
2. *If* $SP = SP_1 \cup SP_2$ *and* $\mathbf{nf}(SP_i) = \langle Symbols(SP_i), \Phi_i, T_i \rangle|_{Sig(SP_i)}$ $i = 1, 2$, *then* $\mathbf{nf}(SP) \stackrel{def}{=} \langle Symbols(SP), in_1(\Phi_1) \cup in_2(\Phi_2), in_1(T_1) \cup in_2(T_2) \rangle|_{Sig(SP)}$ *(see Figure 5.3)*
3. *If* $SP = \textsf{translate } SP' \textsf{ by } \sigma$ *and* $\mathbf{nf}(SP') = \langle Symbols(SP'), \Phi', T' \rangle|_{Sig(SP')}$, *then* $\mathbf{nf}(SP) \stackrel{def}{=} \langle Symbols(SP), \varsigma(\Phi'), \varsigma(T') \rangle|_{Sig(SP)}$ *(see Figure 5.4)*
4. *If* $SP = \textsf{hide sorts } S' \textsf{ opns } F' \textsf{ in } SP'$ *and* $\mathbf{nf}(SP') = \langle Symbols(SP'), \Phi', T' \rangle|_{Sig(SP')}$, *then* $\mathbf{nf}(SP) \stackrel{def}{=} \langle Symbols(SP'), \Phi', T' \rangle|_{Sig(SP)}$

where ς, in_1 *and* in_2 *are extended to translate terms and formulas.*

The class of real models of a specification is equivalent to the class of real models of its corresponding ***nf*** normal form, provided the family of behavioural equalities considered is compatible. In the sequel, let SP be a structured specification and A be a $Sig(SP)$-algebra.

Theorem 5.19 *If* $\approx$ *is compatible, then* $A \in Mod_\approx(SP)$ *if and only if* $A \in Mod_\approx(\mathbf{nf}(SP))$.

Proof. By induction on the structure of SP.

$(i) SP = \langle \Sigma, \Phi, T \rangle$**.** Trivial.

$(ii) SP = SP_1 \cup SP_2$**.** $(\Rightarrow)$ Suppose $A \in Mod_{\approx}(SP)$. Then $A \in Mod_{\approx}(SP_1)$ and $A \in Mod_{\approx}(SP_2)$. By induction hypothesis, $A \in Mod_{\approx}(nf(SP_1))$ and $A \in Mod_{\approx}(nf(SP_2))$, where $nf(SP_i) = \langle \Sigma_i, \Phi_i, T_i \rangle|_{Sig(SP_i)}$, with $\Sigma_i = Symbols(SP_i)$, $i = 1, 2$. By Definition 4.28, $A'_i \in Mod_{\approx}(\langle \Sigma_i, \Phi_i, T_i \rangle)$, that is, $A'_i \models_{\approx_{\Sigma_i}} \Phi_i$, for some Σ_i-algebra A'_i so that $A = A'_i|_{Sig(SP_i)}$. Let $\Sigma = Symbols(SP)$. By the Amalgamation Lemma (see Subsection 2.1.1), there exists a unique Σ-algebra A' such that $A'_i = A'|_{in_i}$ (see Figure 5.3). Also, $A = A'|_{Sig(SP)}$. Because $\approx$ is compatible, then by Theorem 4.12, $A' \models_{\approx_\Sigma} in_i(\Phi_i)$. Then $A' \models_{\approx_\Sigma} in_1(\Phi_1) \cup in_2(\Phi_2)$, and $A' \in Mod_{\approx}(\langle \Sigma, in_1(\Phi_1) \cup in_2(\Phi_2), in_1(T_1) \cup in_2(T_2) \rangle)$. Thus, $A \in Mod_{\approx}(\langle \Sigma, in_1(\Phi_1) \cup in_2(\Phi_2), in_1(T_1) \cup in_2(T_2) \rangle|_{Sig(SP)})$. $(\Leftarrow)$ Similarly.

$(iii) SP =$ translate SP' by σ**, with** $\sigma : Sig(SP') \to Sig(SP)$**.** $(\Rightarrow)$ Suppose $A \in Mod_{\approx}(SP)$. Then, by Definition 4.26, $A|_\sigma \in Mod_{\approx}(SP')$. By induction hypothesis, $A|_\sigma \in Mod_{\approx}(nf(SP'))$, where $nf(SP') = \langle \Sigma', \Phi', T' \rangle|_{Sig(SP')}$, and $\Sigma' = Symbols(SP')$. By Definition 4.28, $A' \in Mod_{\approx}(\langle \Sigma', \Phi', T' \rangle)$, with $A|_\sigma = A'|_{Sig(SP')}$, and so, $A' \models_{\approx_{\Sigma'}} \Phi'$. Let $\Sigma = Symbols(SP)$. By the Amalgamation Lemma, there exists a unique Σ-algebra A'' such that $A' = A''|_\varsigma$ and $A = A''|_{Sig(SP)}$ (see Figure 5.4). By Theorem 4.12, $A'' \models_{\approx_\Sigma} \varsigma(\Phi')$. So, $A'' \in Mod_{\approx}(\langle \Sigma, \varsigma(\Phi'), \varsigma(T') \rangle)$. Thus, $A \in Mod_{\approx}(\langle \Sigma, \varsigma(\Phi'), \varsigma(T') \rangle|_{Sig(SP)})$ by Definition 4.28. $(\Leftarrow)$ Similarly.

$(iv) SP =$ hide sorts S' opns F' in SP'**.** $(\Rightarrow)$ Suppose $A \in Mod_{\approx}(SP)$. Then, by Definition 4.28, $A' \in Mod_{\approx}(SP')$, for some $Sig(SP')$-algebra such that $A = A'|_{Sig(SP)}$. By induction hypothesis, $A' \in Mod_{\approx}(nf(SP'))$, where $nf(SP') = \langle Symbols(SP'), \Phi', T' \rangle|_{Sig(SP')}$. Then, by Definition 4.28, $A'' \in Mod_{\approx}(\langle Symbols(SP'), \Phi', T' \rangle)$, for some $Symbols(SP')$-algebra, such that $A' = A''|_{Sig(SP')}$ and $A = A''|_{Sig(SP)}$. Thus, $A \in Mod_{\approx}(\langle Symbols(SP'), \Phi', T' \rangle|_{Sig(SP)})$ by Definition 4.28. $(\Leftarrow)$ Similarly.

□

5.3.1 Structured Testing and the *nf* Normal Form

Under certain assumptions on the families of equalities, either any checkable model of SP is a checkable model of $\boldsymbol{nf}(SP)$ or vice-versa. It is important to remark that assumptions on quantifiers are due to both the *union* and *hide* operations. Thus, they can be replaced by the assumption that *SP* is *consistent* and *visible w.r.t test sets*. However, the assumptions on quantifiers are required in order to compare the classes of checkable and real models. So, it is better sticking to them in the following theorems.

Theorem 5.20 *If $\sim$ is reduction-compatible, $\simeq$ is translation-compatible and the axioms of SP have only positive occurrences of $\forall$ and negative occurrences of $\exists$, then $A \in ChMod_{\sim,\simeq}(\boldsymbol{nf}(SP))$ implies $A \in ChMod_{\sim,\simeq}(SP)$.*

Proof. By induction on the structure of *SP*. The only interesting case to look at is when $SP = SP_1 \cup SP_2$. Suppose $A \in ChMod_{\sim,\simeq}(\boldsymbol{nf}(SP))$. Then, $A \in ChMod_{\sim,\simeq}(\langle \Sigma, in_1(\Phi_1) \cup in_2(\Phi_2), in_1(T_1) \cup in_2(T_2)\rangle|_{Sig(SP)})$, where $\boldsymbol{nf}(SP_i) = \langle \Sigma_i, \Phi_i, T_i\rangle|_{Sig(SP_i)}$, with $\Sigma_i = Symbols(SP_i)$, $i = 1,2$ and $\Sigma = Symbols(SP)$. By Definition 4.28, $A' \in ChMod_{\sim,\simeq}(\langle \Sigma, in_1(\Phi_1) \cup in_2(\Phi_2), in_1(T_1) \cup in_2(T_2)\rangle)$ for some Σ-algebra A' such that $A = A'|_{Sig(SP)}$. Then $A' \models^{in_1(T_1) \cup in_2(T_2)}_{\sim_\Sigma,\simeq_\Sigma} in_1(\Phi_1) \cup in_2(\Phi_2)$. Because the axioms of *SP* have only positive $\forall$ and negative $\exists$,[2] $A' \models^{in_i(T_i)}_{\sim_\Sigma,\simeq_\Sigma} in_i(\Phi_i)$ by Theorem 3.20. By Theorem 4.9, $A'_i \models^{T_i}_{\sim_{\Sigma_i},\simeq_{\Sigma_i}} \Phi_i$, for some Σ_i-algebra A'_i with $A'_i = A'|_{in_i}$ and $A = A'_i|_{Sig(SP_i)}$. So, $A'_i \in ChMod_{\sim,\simeq}(\langle \Sigma_i, \Phi_i, T_i\rangle)$, and then $A \in ChMod_{\sim,\simeq}(\boldsymbol{nf}(SP_i))$. By induction hypothesis, $A \in ChMod_{\sim,\simeq}(SP_i)$. Hence, $A \in ChMod_{\sim,\simeq}(SP)$. □

Theorem 5.21 *If $\sim$ is translation-compatible, $\simeq$ is reduction-compatible and the axioms of SP have only negative occurrences of $\forall$ and positive occurrences of $\exists$, then $A \in ChMod_{\sim,\simeq}(SP)$ implies $A \in ChMod_{\sim,\simeq}(\boldsymbol{nf}(SP))$.*

Proof. Similar to the proof of Theorem 5.20. □

Obviously, if the families of approximate equalities considered are compatible and *SP* is *consistent* and *visible w.r.t test sets* (assumptions on quantifiers are contradictory here), then the classes of checkable models of *SP* and $\boldsymbol{nf}(SP)$ are equivalent.

Corollary 5.22 *Let SP be consistent and visible w.r.t test sets. If $\sim$ and $\simeq$ are compatible, then $A \in ChMod_{\sim,\simeq}(\boldsymbol{nf}(SP))$ if and only if $A \in ChMod_{\sim,\simeq}(SP)$.*

[2] Note that even if $Test(SP_1) = Test(SP_2)$, $in_1(T_1)$ may differ from $in_2(T_2)$ due to hidden definitions. Thus, $in_i(T_i) \subseteq in_1(T_1) \cup in_2(T_2)$.

Proof. Follows from Theorems 5.20 and 5.21. □

Corollaries 5.23 and 5.24 below relate testing from the ***nf*** normal form and membership in the class of real models. Again, any real model of *SP* is a checkable model of $\boldsymbol{nf}(SP)$, provided the pertinent assumptions are met.

Corollary 5.23 *If $\sim$ is complete, $\doteq$ is sound, $\approx$ is compatible and the axioms of SP have only positive occurrences of $\forall$ and negative occurrences of $\exists$, then $A \in Mod_{\approx}(SP)$ implies $A \in ChMod_{\sim,\doteq}(\boldsymbol{nf}(SP))$.*

Proof. Suppose $A \in Mod_{\approx}(SP)$. By Theorem 5.19, $A \in Mod_{\approx}(\boldsymbol{nf}(SP))$. By Theorem 4.45, $A \in ChMod_{\sim,\doteq}(\boldsymbol{nf}(SP))$. □

Corollary 5.24 *If $\sim$ is sound, $\doteq$ is complete, $\approx$ is compatible and the axioms of SP have only negative occurrences of $\forall$ and positive occurrences of $\exists$, then $A \in ChMod_{\sim,\doteq}(\boldsymbol{nf}(SP))$ implies $A \in Mod_{\approx}(SP)$.*

Proof. Suppose $A \in ChMod_{\sim,\doteq}(\boldsymbol{nf}(SP))$. Then $A \in Mod_{\approx}(\boldsymbol{nf}(SP))$ by Theorem 4.46. By Theorem 5.19, $A \in Mod_{\approx}(SP)$. □

Therefore, from Corollary 5.23 and Theorem 5.20 (Theorem 5.21 and Corollary 5.24), testing from the ***nf*** normal form can be more rigorous than from the original specification (see Figure 5.5). The reason is that test sets are bigger in the former and equality is interpreted in the normal form according to the resulting signature without *export*, whereas equality is interpreted in the original specification according to signature of the subspecification where the axiom being considered belong to. For instance, let $\sigma : \Sigma' \to \Sigma$. If $\sim$ is reduction-compatible, the reduct of $\sim_{\Sigma}$ is either equal to or finer than $\sim_{\Sigma'}$ and if $\doteq$ is translation-compatible, then the reduct of $\doteq_{\Sigma}$ is either equal or coarser than $\doteq_{\Sigma'}$. Nevertheless, according to Figure 5.5, structured testing can also be more rigorous than testing from the ***nf*** normal form, but under more constraining assumptions.

Contrary to the ***con*** normal form, under certain conditions, the class of checkable models of *SP* and $\boldsymbol{nf}(SP)$ are equivalent (see Corollary 5.22). However, the condition that *SP* has to be consistent (or compatible) and visible w.r.t test sets can be very restrictive if one wish to embed the ***nf*** normal into a compositional approach to testing. It can be remarked that if either conditions on equalities or conditions on test

$$
\begin{tikzcd}[column sep=huge, row sep=huge]
A \in Mod_{\approx}(SP) \arrow[rr, Leftrightarrow, "\begin{array}{c}\approx \textit{compatible}\\ \text{Thm.}(5.19)\end{array}"] \arrow[d, Rightarrow, shift right=2, swap, "\begin{array}{c}+\forall\ -\exists\\ \sim \textit{complete}\\ \doteq \textit{sound}\\ \text{Thm.}(4.45)\end{array}"] & & A \in Mod_{\approx}(\boldsymbol{nf}(SP)) \arrow[d, Rightarrow, shift right=2, swap, "\begin{array}{c}+\forall\ -\exists\\ \sim \textit{complete}\\ \doteq \textit{sound}\\ \text{Thm.}(4.45)\end{array}"] \\
A \in ChMod_{\sim,\doteq}(SP) \arrow[u, Rightarrow, shift right=2, swap, "\begin{array}{c}-\forall\ +\exists\\ \sim \textit{sound}\\ \doteq \textit{complete}\\ \text{Thm.}(4.46)\end{array}"] \arrow[rr, Leftrightarrow, "\begin{array}{c}SP\ \textit{consist.+vis.}\\ \sim\ \doteq\ \textit{compat.}\ \text{Cor.}(5.22)\end{array}"] \arrow[rr, Rightarrow, shift right=6, swap, "\begin{array}{c}-\forall\ +\exists\\ \sim \textit{tran.compat.}\\ \doteq \textit{red.compat.}\\ \text{Thm.}(5.21)\end{array}"] & & A \in ChMod_{\sim,\doteq}(\boldsymbol{nf}(SP)) \arrow[u, Rightarrow, shift right=2, swap, "\begin{array}{c}-\forall\ +\exists\\ \sim \textit{sound}\\ \doteq \textit{complete}\\ \text{Thm.}(4.46)\end{array}"] \arrow[ll, Rightarrow, shift right=6, swap, "\begin{array}{c}+\forall\ -\exists\\ \sim \textit{red.compat.}\\ \doteq \textit{tran.compat.}\\ \text{Thm.}(5.20)\end{array}"]
\end{tikzcd}
$$

Figure 5.5: Testing from the ***nf*** normal form, structured testing and correctness.

sets (quantifiers) are not met, we have no result relating testing from the ***nf*** normal form to testing from the original specification.

***Example* 5.25** Consider again the *IntList* specification presented in Example 4.22. From Definition 5.18, the normal form $\boldsymbol{nf}(IntList)$ is as follows.

$$\boldsymbol{nf}(IntList) = \langle \Sigma_2, \Phi_1 \cup \Phi_2, T_1 \cup T_2 \rangle|_{Sig(IntList)}$$

where Φ_1 is the set of axioms of *List*, Φ_2 is the set of axioms of *SList*2 and $Symbols(IntList) = Symbols(SList) = \sigma(\Sigma_1) +_{\Sigma_2} \Sigma_2 = \Sigma_2$, since no hidden symbols need to be renamed. Let A be an *IntList*-algebra. Structured testing from the normal form consists in a single test $A' \models^{T_1 \cup T_2}_{\sim_{\Sigma_2}, \doteq_{\Sigma_2}} \Phi_1 \cup \Phi_2$ with $A = A'|_{Sig(IntList)}$. Concerning quantifiers, the *IntList* specification matches Corollary 5.23 and Theorem 5.20. So, in order to test from the normal form instead of the original specification the following must be met. From Corollary 5.23, $\approx$ must be compatible. Suppose $\approx$ is the family of observational equalities. This family is compatible when focusing on *IntList*, notwithstanding *is_sorted* which is a *list* observer is added in Σ_2 and not in Σ_1 and also is subsequently hidden. Furthermore, $\sim$ needs to be complete and reduction-compatible and $\doteq$ needs to be sound and translation-compatible. It is easy to check that any $\sim$ defined from a finite subset of observable contexts is both complete w.r.t. $\approx$ and reduction-compatible. Also, the literal equality = is both sound and translation-compatible. $\square$

Example* 5.26 (Unification *continued*)** Consider Example 4.61. The ***nf normal form of the UNIFICATION specification groups hidden definitions and reduces the number of tests to be performed, although the family of behavioural equalities is now required to be compatible. But, notice that, if $\approx$ is the family of observational equalities, then it is compatible when focusing on the signatures arising in the structure of UNIFICATION. Finally, the families of approximate equalities defined there, namely $\overline{\sim}$ and $\simeq$, are complete and reduction-compatible, and sound and translation-compatible respectively. It is easy to check that testing from the ***nf*** normal form can be more rigorous than pure structured testing, even though they may not be equivalent, unless both $\overline{\sim}$ and $\simeq$ are compatible and UNIFICATION is consistent w.r.t test sets. $\square$

5.3.2 Flat Testing and the *nf* Normal Form

Due to the ***nf*** normal form, the compatibility condition on families of approximate equalities can be weakened for flat testing. However, the family of behavioural equalities $\approx$ is required to be compatible. In fact, testing from the ***nf*** normal form is basically flat testing from the unstructured set of all specification axioms including the hidden ones. In addition, the consistency condition on test sets in Theorem 4.51 can be dropped altogether, provided that $\approx$ is compatible (see Figures 4.1 and 5.6). The following is a corollary of Theorems 5.19, 4.45 and 4.9 and an alternative to Theorem 4.51.

Corollary 5.27 *If $\sim$ is complete and reduction-compatible and $\doteq$ is sound and translation-compatible, $\approx$ is compatible and the axioms of SP have only positive occurrences of $\forall$ and negative occurrences of $\exists$, then $A \in Mod_{\approx}(SP)$ implies $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$.*

Proof. Suppose $A \in Mod_{\approx}(SP)$. Then, by Theorem 5.19, $A \in Mod_{\approx}(nf(SP))$. By Theorem 4.45, $A \in ChMod_{\sim,\doteq}(nf(SP))$. Then, by Definition 4.28 and 5.18, $A' \models^{T}_{\sim_{Symbols(SP)},\doteq_{Symbols(SP)}} \Phi$, where $nf(SP) = \langle Symbols(SP), \Phi, T\rangle|_{Sig(SP)}$ and $A = A'|_{Sig(SP)}$. Because $Test(SP) \subseteq T$ and $TAx(SP) \subseteq \Phi$ and the axioms of SP have only positive $\forall$ and negative $\exists$, then $A' \models^{Test(SP)}_{\sim_{Symbols(SP)},\doteq_{Symbols(SP)}} TAx(SP)$. By Theorem 4.9, $A \models^{Test(SP)}_{\sim_{Sig(SP)},\doteq_{Sig(SP)}} TAx(SP)$. $\square$

The dual of Corollary 5.27 has a similar proof.

Corollary 5.28 *Let SP be transparent. If $\sim$ is sound and translation-compatible and $\doteq$ is complete and reduction-compatible, $\approx$ is compatible and the axioms of SP have only negative occurrences of $\forall$ and positive occurrences of $\exists$, then $A \models^{Test(SP)}_{\sim,\doteq} TAx(SP)$ implies $A \in Mod_{\approx}(SP)$.*

Proof. Suppose $A \models^{Test(SP)}_{\sim_{Sig(SP)},\doteq_{Sig(SP)}} TAx(SP)$. Then $A' \models^{Test(SP)}_{\sim_{Symbols(SP)},\doteq_{Symbols(SP)}} TAx(SP)$ by Theorem 4.10, where $A = A'|_{Sig(SP)}$ and $\boldsymbol{nf}(SP) = \langle Symbols(SP), \Phi, T\rangle|_{Sig(SP)}$. Because $Test(SP) = T$ and $TAx(SP) = \Phi$ (SP is transparent), then $A \in ChMod_{\sim,\doteq}(\boldsymbol{nf}(SP))$. By Theorem 4.46, $A' \in Mod_{\approx}(\boldsymbol{nf}(SP))$. Hence, by Theorem 5.19, $A \in Mod_{\approx}(SP)$. $\square$

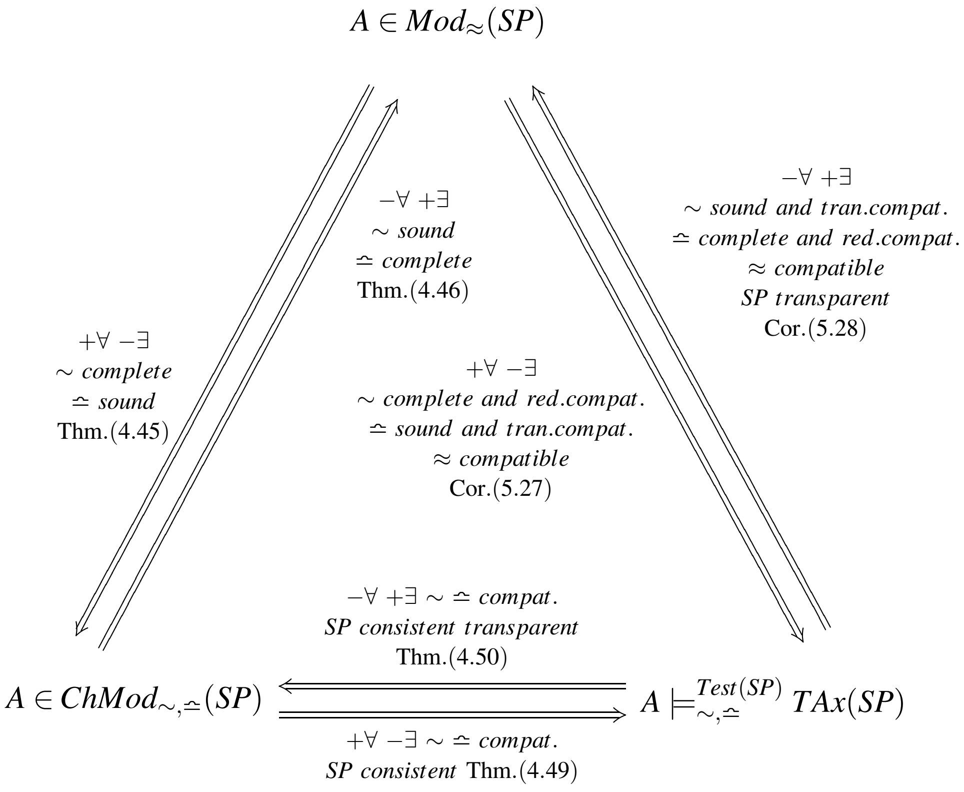

Figure 5.6: Structured Testing, flat testing, correctness and the ***nf*** normal form.

Example* 5.29** In Example 4.22, testing from the ***nf normal form is similar to flat testing, except from the fact that, in the former, hidden symbols and axioms are also taken into account and the test set is not restricted to the signature of $IntList$. As mentioned before, when focusing on $IntList$, the family of observational equalities is compatible. Moreover, since the compatibility condition on families of approximate equalities is weakened by the ***nf*** normal form, there is more flexibility to define families of equalities either testing from the normal form (Corollary 5.23) or by flat testing (Corollary 5.27). Nevertheless, notice that, for flat testing, one could chose between fulfilling the assumptions of either Theorem 4.51 (Corollary 5.13) or Corollary 5.27. In case of Example 4.22, it is better relying on Corollary 5.27. But, suppose the following observer is added to Σ_2: $ident : list \rightarrow nat$, which returns the identifier of a list. Then, the family of observational equalities is no longer compatible. By relying on Theorem 4.51, one could define a family of approximate equalities which is complete and compatible, by taking only observers in Σ_1 into account, without needing to change the specification. Furthermore, the main advantage of testing from the normal form over flat testing is to make it possible for hidden axioms to be checked and test sets may be bigger since hidden terms are also considered. □

Example* 5.30 (Unification *continued*)** Due to the ***nf normal form, flat testing in Example 4.62 can be performed with more precise equalities, since they are no longer required to be compatible, but the first needs to be reduction-compatible and the second needs to be translation-compatible. Thus, equalities defined in Example 4.61 can be used instead of the ones in Example 4.62. Also, because $\approx$ is compatible, Corollary 5.27 can be considered, which means that UNIFICATION is no longer required to be either consistent or compatible w.r.t. test sets. □

5.4 The *snf* Normal Form

The strict normal form (***snf***) is similar to the ***nf*** normal form, but aimed at achieving the main benefits of the ***con*** and ***nf*** normal forms by assuring equivalence of model classes while completely eliminating the consistency w.r.t test sets requirement of non-compositional approaches. In order to drop this consistency condition, it is necessary to make sure axioms are interpreted according to the test set given in the basic specification they belong to. Strict specifications are flat specifications where each axiom is associated with a test set.

Definition 5.31 (Strict Specification) *A* strict *specification* $SP = \langle \Sigma, \Psi \rangle$ *with* $\Psi \subseteq \{(\psi, T) \mid \psi \in Sen(\Sigma) \text{ and } T \subseteq T_\Sigma\}$ *is defined as follows.*

- $Sig(SP) \stackrel{def}{=} \Sigma$
- $Test(SP) \stackrel{def}{=} (\bigcup_{(\psi,T)\in\Psi} T)$
- $Mod_\approx(SP) \stackrel{def}{=} \{A \in Alg(\Sigma) \mid \bigwedge_{(\psi,T)\in\Psi} A \models_\approx \psi\}$
- $ChMod_{\sim,\doteq}(SP) \stackrel{def}{=} \{A \in Alg(\Sigma) \mid \bigwedge_{(\psi,T)\in\Psi} A \models^T_{\sim,\doteq} \psi\}$

The strict normal form (***snf***) of a specification SP is a strict specification restricted by the export operator given in Section 5.3. Since strict specifications are regarded as structured specifications, the export operator is also applicable here.

Definition 5.32 (*snf* Normal Form) *Let SP be a structured specification. The strict normal form* $\textbf{snf}(SP)$ *is defined as follows.*

1. *If* $SP = \langle \Sigma, \Phi, T \rangle$*, then* $\textbf{snf}(SP) \stackrel{def}{=} \langle \Sigma, \Psi \rangle|_\Sigma$*, where* $\Psi = \{(\phi, T) \mid \phi \in \Phi\}$
2. *If* $SP = SP_1 \cup SP_2$ *and* $\textbf{snf}(SP_i) = \langle Symbols(SP_i), \Psi_i \rangle|_{Sig(SP_i)}$ $i = 1, 2,$ *then* $\textbf{snf}(SP) \stackrel{def}{=} \langle Symbols(SP), in_1(\Psi_1) \cup in_2(\Psi_2) \rangle|_{Sig(SP)}$ *(see Figure 5.3)*
3. *If* $SP =$ translate SP' by σ *and* $\textbf{snf}(SP') = \langle Symbols(SP'), \Psi' \rangle|_{Sig(SP')}$*, then* $\textbf{snf}(SP) \stackrel{def}{=} \langle Symbols(SP), \varsigma(\Psi') \rangle|_{Sig(SP)}$ *(see Figure 5.4)*
4. *If* $SP =$ hide sorts S' opns F' in SP' *and*
   $\textbf{snf}(SP') = \langle Symbols(SP'), \Psi' \rangle|_{Sig(SP')}$*,*
   then $\textbf{snf}(SP) \stackrel{def}{=} \langle Symbols(SP'), \Psi' \rangle|_{Sig(SP)}$

where ς*,* in_1 *and* in_2 *are extended to translate pairs of formulas and terms.*

The classes of real models of SP and ***snf***(SP) are equivalent whenever the family of behavioural equalities considered is compatible. In the sequel, let SP be a structured specification and A be a $Sig(SP)$-algebra.

Theorem 5.33 *If* $\approx$ *is compatible, then* $A \in Mod_\approx(SP)$ *if and only if* $A \in Mod_\approx(\textbf{snf}(SP))$.

Proof. Similar to the proof of Theorem 5.19. □

Correspondingly, the classes of checkable models of SP and $\boldsymbol{snf}(SP)$ are equivalent if the families of approximate equalities are compatible. Yet this condition can be weakened if one is only interested in one or the other direction of the implication. Notice that, contrary to the ***nf*** normal form, neither conditions on test sets nor conditions on axioms are necessary.

Theorem 5.34 *If $\sim$ is reduction-compatible and $\simeq$ is translation-compatible, then $A \in ChMod_{\sim,\simeq}(\boldsymbol{snf}(SP))$ implies $A \in ChMod_{\sim,\simeq}(SP)$.*

Proof. By induction of the structure of SP. The only interesting case to look at is when $SP = SP_1 \cup SP_2$. Suppose $A \in ChMod_{\sim,\simeq}(\boldsymbol{snf}(SP))$. Then $A \in ChMod_{\sim,\simeq}(\langle \Sigma, in_1(\Psi_1) \cup in_2(\Psi_2)\rangle|_{Sig(SP)})$, where $\Sigma = Symbols(SP)$ and $\boldsymbol{snf}(SP_i) = \langle \Sigma_i, \Psi_i\rangle|_{Sig(SP_i)}$ with $\Sigma_i = Symbols(SP_i)$, $i = 1,2$. So, by Definition 4.28, $A' \in ChMod_{\sim,\simeq}(\langle \Sigma, in_1(\Psi_1) \cup in_2(\Psi_2)\rangle)$, that is, $\bigwedge_{(\psi,T)\in in_1(\Psi_1)\cup in_2(\Psi_2)} A' \models^T_{\sim_\Sigma,\simeq_\Sigma} \psi$, for some Σ-algebra A' with $A = A'|_{Sig(SP)}$. Then, obviously, $\bigwedge_{(\psi,T)\in in_i(\Psi_i)} A' \models^T_{\sim_\Sigma,\simeq_\Sigma} \psi$. By Theorem 4.9, $\bigwedge_{(\psi,T)\in\Psi_i} A'_i \models^T_{\sim_{\Sigma_i},\simeq_{\Sigma_i}} \psi$, for some Σ_i-algebra A'_i so that $A'_i = A'|_{in_i}$ and $A = A'_i|_{Sig(SP_i)}$. So, $A'_i \in ChMod_{\sim,\simeq}(\langle \Sigma_i, \Psi_i\rangle)$. By Definition 4.28, $A \in ChMod_{\sim,\simeq}(\boldsymbol{snf}(SP_i))$. By induction hypothesis, $A \in ChMod_{\sim,\simeq}(SP_i)$. Hence, $A \in ChMod_{\sim,\simeq}(SP)$. □

Theorem 5.35 *If $\sim$ is translation-compatible and $\simeq$ is reduction-compatible, then $A \in ChMod_{\sim,\simeq}(SP)$ implies $A \in ChMod_{\sim,\simeq}(\boldsymbol{snf}(SP))$.*

Proof. Similar to the proof of Theorem 5.34. □

The following corollary shows that we can always substitute the strict normal of SP for SP if the approximate equalities are compatible.

Corollary 5.36 *If $\sim$ and $\simeq$ are compatible, then $A \in ChMod_{\sim,\simeq}(\boldsymbol{snf}(SP))$ if and only if $A \in ChMod_{\sim,\simeq}(SP)$.*

Proof. Follows from Theorems 5.34 and 5.35. □

Similar to the ***nf*** normal form, testing from the ***snf*** normal form can be more rigorous than from the original specification if certain conditions are met (see Corollary 5.37 and Theorem 5.34).

Corollary 5.37 *If $\sim$ is complete, $\doteq$ is sound, $\approx$ is compatible and the axioms of SP have only positive occurrences of $\forall$ and negative occurrences of $\exists$, then $A \in Mod_{\approx}(SP)$ implies $A \in ChMod_{\sim,\doteq}(\boldsymbol{snf}(SP))$.*

Proof. Suppose $A \in Mod_{\approx}(SP)$. By Theorem 5.33, $A \in Mod_{\approx}(\boldsymbol{snf}(SP))$. By Theorem 4.45[3], $A \in ChMod_{\sim,\doteq}(\boldsymbol{snf}(SP))$. □

Corollary 5.38 below is the dual of Corollary 5.37.

Corollary 5.38 *If $\sim$ is sound, $\doteq$ is complete, $\approx$ is compatible and the axioms of SP have only negative occurrences of $\forall$ and positive occurrences of $\exists$, then $A \in ChMod_{\sim,\doteq}(\boldsymbol{snf}(SP))$ implies $A \in Mod_{\approx}(SP)$.*

Proof. Suppose $A \in ChMod_{\sim,\doteq}(\boldsymbol{snf}(SP))$. Then $A \in Mod_{\approx}(\boldsymbol{snf}(SP))$ by Theorem 4.46. Thus, by Theorem 5.33, $A \in Mod_{\approx}(SP)$. □

Furthermore, as one might expect, ***snf*** normal forms can always be incremented by additional test sets (according to Definition 5.1), but equivalence of the classes of checkable models can be lost.

***Example* 5.39** Consider again the *IntList* specification presented in Example 4.22. From Definition 5.32, the normal form $\boldsymbol{snf}(IntList)$ is as follows.

$$\boldsymbol{snf}(IntList) = \langle \Sigma_2, \Psi_1 \cup \Psi_2 \rangle|_{Sig(IntList)}$$

where $\Psi_i = \{(\phi_i, T_i) \mid \phi_i \in \Phi_i\}$ with $i = 1,2$, Φ_1 is the set of axioms of *List*, Φ_2 is the set of axioms of *SList*2 and $Symbols(IntList) = \Sigma_2$ as in Example 5.25. Let A be an *IntList*-algebra. Structured testing from the normal form corresponds to $\bigwedge_{(\psi,T)\in\Psi_1\cup\Psi_2} A' \models^{T}_{\sim_{\Sigma_2},\doteq_{\Sigma_2}} \psi$ with $A = A'|_{Sig(IntList)}$. Like structured testing, hidden symbols are taken into account and each axiom is checked with the original test set associated with it. On the other hand, like flat testing, axioms are grouped and equalities on a single signature, in this case Σ_2, are used. Notice that *IntList* matches Corollary 5.37. The family of observational equalities, $\approx$, needs to be compatible, which is the case. Finally, a complete and reduction-compatible and a sound and translation-compatible family of equalities need to be defined (see Example 5.25). Let $\sim$ and $\doteq$ be defined as in Subsection 4.6.1. In this case, it is easy to check that testing from the ***snf*** normal form is more rigorous than testing from the original specification, since $\sim_{\Sigma_2}$ (which includes *is_sorted* as observer) is finer than $\sim_{\Sigma_1}$. □

[3] Although strict specifications are not considered in Theorem 4.45, it is obvious that this theorem can be extended to consider them without further constraints. The same applies to Theorem 4.46.

The main advantage of the ***snf*** normal form is to allow a combination of compositional and non-compositional testing, namely semi-structured testing, which is more likely to be adopted in practice. For instance, ***snf*** normal forms can be constructed for replacing some parts of a specification, especially when these parts are combined by *union*. Then, the resulting specification can be checked by structured testing. For this, strict specifications are regarded as a new form of structured specifications. One drawback of this combined approach is that compatibility of equalities which is not required by pure structured testing (see Theorems 4.45 and 4.46), has to be met for both behavioural and approximate equalities (see Theorem 5.33 and Corollary 5.36). How to weaken this condition still needs to be investigated. As mentioned in Section 4.6, the family of observational equalities $\approx$ may not be translation-compatible, unless the specification preserves encapsulation. Concerning test sets, testing from the ***snf*** normal form is similar to pure structured testing in the sense that they precisely comply with the testing interface. This is not the case when the ***nf*** and ***con*** normal forms are considered.

To summarise, Figure 5.7 illustrates the relationship between testing against SP and $\boldsymbol{snf}(SP)$ and correctness.

$$\begin{tikzcd}[column sep=huge, row sep=huge]
A \in Mod_{\approx}(SP) \arrow[r, Leftrightarrow, "\begin{array}{c}\approx\ compatible\\ \text{Thm.}(5.33)\end{array}"] \arrow[d, Rightarrow, shift right, "\begin{array}{c}+\forall\ -\exists\\ \sim complete\\ \simeq sound\\ \text{Thm.}(4.45)\end{array}"'] & A \in Mod_{\approx}(\boldsymbol{snf}(SP)) \arrow[d, Rightarrow, shift right, "\begin{array}{c}+\forall\ -\exists\\ \sim complete\\ \simeq sound\\ \text{Thm.}(4.45)\end{array}"'] \\
A \in ChMod_{\sim,\simeq}(SP) \arrow[u, Rightarrow, shift right, "\begin{array}{c}-\forall\ +\exists\\ \sim sound\\ \simeq complete\\ \text{Thm.}(4.46)\end{array}"'] \arrow[r, Leftrightarrow, "\sim\simeq compat.\ \text{Cor.}(5.36)"] \arrow[r, Rightarrow, shift right=3, "\begin{array}{c}\sim tran.compat.\\ \simeq red.compat.\\ \text{Thm.}(5.35)\end{array}"'] & A \in ChMod_{\sim,\simeq}(\boldsymbol{snf}(SP)) \arrow[u, Rightarrow, shift right, "\begin{array}{c}-\forall\ +\exists\\ \sim sound\\ \simeq complete\\ \text{Thm.}(4.46)\end{array}"'] \arrow[l, Rightarrow, shift right=3, "\begin{array}{c}\sim red.compat.\\ \simeq tran.compat.\\ \text{Thm.}(5.34)\end{array}"']
\end{tikzcd}$$

Figure 5.7: Testing from the ***snf*** normal form, structured testing and correctness.

Example* 5.40 (Unification *continued*)** Because the ***snf normal form preserves model class equivalence if families of equalities are compatible, without constraints on test sets, this form can be used to replace subspecifications. Consider again Example 4.61. In order to reduce the number of tests to be performed and group hidden definitions, the SUBSTITUTION specification can be replaced by the corresponding ***snf*** normal form, since, when focusing on the signatures arising in SUBSTITUTION, the family of behavioural equalities is compatible and the families of approximate equalities defined in Example 4.61 are also compatible (see Corollaries 5.36 and 5.37). Then structured testing can be performed on the modified specification. □

5.5 Concluding Remarks

Normalisation has been regarded as an approach for defining non-compositional proof systems for structured specifications such that theorems can be derived in the underlying institution. In this chapter, normalisation is presented as a way of simplifying testing from structured specifications and handling the oracle problem. It is clear that pure compositional approaches can be inefficient in practice and non-compositional approaches are unstructured and limited by the oracle problem. Flat testing can noticeably benefit from normal forms, since the conditions under which results are satisfactory become weaker. Concerning structured testing, a combined approach in which parts of the specification are replaced by normal forms and the overall specification is checked by structured testing seems to be more promising. For instance, the number of experiments necessary can be reduced, mainly when the specification has several occurrences of *union*. Also, grouping hidden definitions can help to systematise an implementation of hidden sorts which can be a difficult task, but it is essential to test interesting properties expressed by hidden axioms when they cannot be replaced by their visible consequences. Moreover, tests from normal forms can be more rigorous than from the original specifications. This can be clearly achieved by applying the ***inc*** operation and, consequently, the ***con*** normal form. Regarding the ***nf*** and ***snf*** normal forms, if the complete (and the sound) equality for $Symbols(SP)$ whose family is required to be reduction-compatible (translation-compatible) is finer (coarser) than the ones for the signatures of the parts of *SP*, than testing from the normal forms can be more rigorous as well. Examples of such reduction-compatible (translation-compatible) families of approximate equalities which are complete (sound) w.r.t. a compatible family of behavioural equalities together with how this compatibility condition on approximate and behavioural equalities can be weakened are object of further investigation. Finally, the classes of real models of the normal forms and the original specification are equivalent (provided that the family of behavioural equalities considered is compatible, in the case of ***nf*** and ***snf***).

Chapter 6

Conclusions and Further Work

"Why does this magnificent applied science which saves work and makes life easier bring us so little happiness ? The simple answer runs: because we have not yet learned to make sensible use of it."
A. Einstein

Contents

The oracle problem for testing programs against structured algebraic specifications expressed in first-order logic is the main subject of this thesis. This problem is systematically investigated and a solution is proposed focusing on flat specifications and then extending to structured specifications. A primary contribution of this thesis to the general area of specification-based testing and formal methods is to provide a basis for an understanding of several issues related to the oracle problem in contexts where functionality and modularity are fundamental. Theoretical foundations for interpreting test results against structured algebraic specifications are given and also refined and extended from previous work in the area. Some obstacles which may impede the definition of oracles are identified together with some possible ways of overcoming them. Whether success (failure) in testing means correctness (incorrectness) can be detected under certain constraints. But, it is likely that, in many cases, only a partial guarantee can be achieved.

The next sections present the main contributions of this thesis highlighting main benefits, problems, what can be improved along with some further work.

6.1 Approximate Oracles

The oracle problem for flat specifications is often caused by the equality problem – the question of how equality on non-observable sorts can be defined – and the quantifier problem – many quantifiers range over domains that can be infinite. The equality problem can be handled by the use of behavioural equalities and, in particular, observational equalities, whereas the quantifier problem can be handled by choosing valid and unbiased test sets. However, this solution is not always feasible (see Section 3.1). This thesis presents a solution to the oracle problem based on the use of approximate equalities to define approximate oracles and taking the syntactic position of quantifiers into account to handle the quantifier problem. The intention is to define two equalities which approximate behavioural equality from opposite directions – one sound and one complete – and apply them to compare values of a non-observable sort according to the syntactic occurrence of equations in axioms, depending on whether an occurrence is positive or negative (see Sections 3.2 and 3.3). Behavioural equalities are difficult or even impossible to define in practice, whereas approximate equalities are much more practical and easier to define since they need not even be congruence relations. The solution can be applied in a wider context than the one presented in [Gaudel, 1995] and it is also less restrictive. Also, it covers a prevalent use of quantifiers, where there are only positive occurrences of $\forall$ and negative occurrences of $\exists$. Approximate oracles can be defined independently of test sets. However, it does not cover the case when both quantifiers are present in a positive position.

Valid and unbiased test sets can be a solution to the quantifier problem when both positive and negative occurrences of quantifiers are present. However, they are difficult to define and normally they are infinite. An automatic method of defining valid and unbiased test sets is unlikely to exist. There is a chance that test sets can be refined towards valid and unbiased ones by adjusting approximate equalities (see Section 3.5). However, there is still no concrete evidence that this can be achieved in practice. It is important to remark that test sets can always be replaced by supersets without losing the validity and/or unbiased property, provided that assumptions on quantifiers are made (see Theorem 3.20). Also, the completeness and soundness of approximate equalities are not affected by refining test sets. On the other hand, even though it may be interesting to refine approximate equalities towards behavioural equalities, there is no guarantee that completeness or soundness is maintained and also test sets may become invalid or biased. Techniques for selecting and refining test sets are still to be defined/improved. Some approaches to select test sets are based on extracting cases from formal proofs [Bernot et al., 1991, Le Gall, 1999] and results obtained by model checking [Ammann et al., 1998, Bousquet, 1999]. A further solution to the quantifier problem can be based on refining axioms using techniques already proposed for selecting test cases based on normalisation of axioms and rewriting [Antoy and Gannon, 1994, Stepney, 1995, Donat, 1997].

An approach to defining approximate oracles – the grey-box approach – is proposed in this thesis. This approach combines the black-box and white-box approaches presented in Section 3.1 to define approximate oracles where the black-box approach is used to define a complete equality from a finite subset of observable contexts and the white-box approach is used to produce a sound equality based on the concrete representation of values. Any contextual equality is complete, whereas structural equalities based on the equality of the values of the concrete representation are always sound. As illustrated in Example 3.30, the grey-box approach seems to be more flexible and reasonable than the white-box and black-box ones. But, a general method of applying the grey-box approach still needs to be defined. Tool support is also vital to automate and assist the definition of oracles.

Although the issue of functionality for flat specifications considered in this thesis is an important one, there are other concerns related to the oracle problem like partiality, exceptions, termination, non-determinism and timing, which are also important when more complex formalisms are considered. These are object of further investigation.

6.2 Structured Testing

The oracle problem for testing from structured specifications reduces to the equality and quantifier problems when different signatures and specification-building operations in the structure are involved and also the problem caused by hidden symbols (see Section 4.1). Structured testing is a style of testing presented in this thesis which is based on the compositional semantics of specifications and copes with the oracle problem by taking the structure of specifications into account in order to define the tests to be performed and also to make sense of axioms. Specific test sets and equalities are applied for specific groups of axioms according to the specification and an additional implementation of hidden symbols is demanded which is vital if hidden axioms need to be checked. Interesting properties of visible functions may be expressed by hidden axioms (see Examples 4.22 and 3.30). Previous results on the use of approximate oracles obtained in Section 3.3 (Section 4.3) are extended to structured testing without further restrictions (see Section 4.5). In this sense, structured testing is more flexible than non-compositional approaches like flat testing. Tests are also easier to plan and maintain. However, the problem of giving an implementation of hidden symbols is a major one and other ways of weakening it should be worked out. Tool assistance would also be handy to automate the process of defining the necessary tests to be performed and to support oracle design, implementation of hidden symbols and test execution (see Section 4.7). Moreover, it would also be interesting to extend CASL to support specifications with testing interface. Furthermore, testing from generic and architectural specifications deserves further investigation.

6.3 Flat Testing

One way of dealing with hidden definitions is to ignore them altogether. Flat testing is a style of testing which checks a program against an unstructured set of visible axioms, namely the set of testable axioms (see Definition 4.20), computed from the original specification. In this case, a single set of tests is performed considering a single pair of approximate equalities on the resulting signature of the specification together with a single test set also computed from the specification. Due to this fact the oracle problem becomes more difficult (see Section 4.1) and, in order to sort it out, flat testing is restricted to specifications which are consistent w.r.t test sets and families of approximate equalities are required to be compatible with signature morphisms (see Sections 4.4.4 and 4.5). In the observational case, the latter condition means that specifications have to preserve encapsulation (see Section 4.6). Families of contextual equalities which are reduction-compatible are easy to define, but they may not be translation-compatible. Moreover, the consistency condition may impede certain tests from being performed (see Section 4.5). In addition, specifications may be required to be transparent, that is, *hide* is completely excluded, which can be too restrictive. Furthermore, structured testing and flat testing are not equivalent, insofar as contradictory assumptions on quantifiers are necessary. Nevertheless, if only specifications which are visible w.r.t test sets are considered, then they are equivalent under the assumptions that specifications are also consistent w.r.t. test sets and transparent and the families of approximate equalities considered are compatible. Therefore, flat testing is more limited than structured testing, not mention the lack of structure in the experiment itself. Furthermore, the process of selecting visible axioms can be fully automated and assistant tools for oracle design and to guide the testing process can be very helpful.

6.4 Normalisation

Normalisation of specifications with testing interface is used in this thesis to deal with the oracle problem for testing from structured specifications. Three normal forms are presented with specific motivations and objectives. The implications on the use of normal forms to the oracle problem is discussed.

The ***con*** normal form is aimed at weakening the consistency condition on test sets when interpreting flat testing. Axioms can be checked with different test sets due to *union*, *translate* and *hide*. The ***con*** normal form increments test sets at each occurrence of *union* without changing the structure of the specification. If the specification is compatible w.r.t. test sets, the ***con*** normal form produces a new specification which is consistent w.r.t. test sets. Obviously, the class of real models of a specification SP and $\textbf{\textit{con}}(SP)$ are equivalent, whereas the class of checkable models are not equivalent in general. Structured testing can also benefit from the ***con*** normal form by making the testing experiment be more rigorous.

The ***nf*** normal form extends the normal form presented in [Bidoit et al., 1999] for ordinary specifications. The intention is to group axioms in order to handle the complexity of structured specifications and deal with hidden definitions. The result is a flat specification which exports visible symbols. The class of real models of a specification SP and the corresponding normal form $\textbf{\textit{nf}}(SP)$ are equivalent provided that the family of behavioural equalities considered is compatible with signature morphisms. Also, the class of checkable models of SP and $\textbf{\textit{nf}}(SP)$ are equivalent if the families of approximate equalities are compatible with signature morphisms and SP is consistent and visible w.r.t. test sets. However, testing from the ***nf*** normal form can be more rigorous than from the original specification, since test sets are bigger and equality is interpreted according to the signature of the specification. From Corollary 5.23 and Theorem 5.20 the family $\sim$ is required to be complete and reduction-compatible. Complete families of equality consistently defined are usually reduction-compatible (see Section 4.6). But the compatibility condition on the family of behavioural equalities is a major one. For flat testing, the compatibility condition of families of approximate equalities can be weakened for flat testing. Also, the consistent condition on test sets can dropped. However, once more the family of behavioural equalities is required to be compatible. Practical ways of weakening this condition still needs to be devised.

The ***snf*** normal form is aimed at assuring equivalence of classes of checkable models and also eliminating the consistency condition on test sets of non-compositional approaches. The idea is that axioms are associated with the test set given in the basic specification they belong to. The class of real models of a specification *SP* and the corresponding normal form ***snf***(SP) are equivalent if the family of behavioural equalities considered is compatible w.r.t signature morphisms. Also, the class of checkable models of *SP* and ***snf***(SP) are equivalent if the families of approximate equalities are compatible. Notice, however, that, contrary to the ***nf*** normal form, conditions on test sets or quantifiers are not necessary. The reason is that testing from the ***snf*** normal form is closer to structured testing than any other non-compositional approach presented in this thesis.

A combination of compositional and non-compositional testing seems more likely to be adopted in practice. For instance, ***snf*** normal forms can replace some parts of the specification (mainly when the compatibility condition of the family of behavioural equalities is not a problem) and then the resulting specification can be checked by structured testing (see Example 5.40). This is called semi-structured testing. A method of applying it still needs to be defined. Moreover, regarding the ***nf*** and ***snf*** normal forms, since hidden symbols are grouped, the problem of implementing hidden symbols can be sorted out by considering only the symbols which can be implemented in cost-effective way. Nevertheless, the application of normal forms can be restricted due to the compatibility condition on the family of behavioural equalities and also the loss of structure which can be crucial for large specifications. Tool assistance to compute normal forms can be very helpful.

Furthermore, a combination and a cooperation of testing, model checking and formal proofs to verify systems in a formal development framework seem to be very promising. For this, testing can be planned and applied at different stages and from the beginning of the development process. These are also object of further investigation.

Bibliography

[Ammann et al., 1998] Ammann, P. E., Black, P. E., and Majurisk, W. (1998). Using model checking to generate tests from specifications. In *Proceedings of Second IEEE International Conference on Formal Engineering Methods - ICFEM '98*.

[Antoy and Gannon, 1994] Antoy, S. and Gannon, J. (1994). Using term rewriting systems to verify software. *IEEE Transactions on Software Engineering*, 20(4):259–274.

[Arnould et al., 1996] Arnould, A., Le Gall, P., and Marre, B. (1996). Dynamic testing from bounded data type specifications. In Hlawiczka, A., Silva, J. G. S., and Simoncini, L., editors, *Second European Dependable Computing Conferences - EDCC-2*, volume 1150 of *Lecture Notes in Computer Science*, Taormina, Italy. Springer.

[Astesiano et al., 1999] Astesiano, E., Kreowski, H.-J., and Krieg-Brückner, B., editors (1999). *Algebraic Foundations of Systems Specifications*. IFIP State-of-The-Art Reports. Springer.

[Barbey et al., 1996] Barbey, S., Buchs, D., and Péraire, C. (1996). A theory of specification-based testing for object-oriented software. In *Second European Dependable Computing Conference - EDCC-2*, volume 1150 of *Lecture Notes in Computer Science*, pages 303–320, Taormina, Italy. Springer.

[Bergstra et al., 1990] Bergstra, J. A., Heering, J., and Klint, P. (1990). Module algebra. *Journal of the Association for Computing Machinery*, 37(2):335–372.

[Bernot, 1989] Bernot, G. (1989). A formalism for test with oracle based on algebraic specifications. Technical Report 89-4, LIENS/DMI, Ecole Normale Supérieure, Paris, France.

[Bernot, 1991] Bernot, G. (1991). Testing against formal specifications: a theoretical view. In Abramsky, S. and Maibaum, T., editors, *Proceedings of the International*

Joint Conference on Theory and Practice of Software Development - TAPSOFT'91, volume 494 of *Lecture Notes in Computer Science*, Brighton, UK. Springer.

[Bernot et al., 1997] Bernot, G., Bouaziz, L., and Le Gall, P. (1997). A theory of probabilistic functional testing. In *Proceedings of International Conference on Software Engineering - ICSE'97*, pages 216–226, Boston MA, USA. ACM.

[Bernot et al., 1991] Bernot, G., Gaudel, M.-C., and Marre, B. (1991). Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405.

[Bertolino and Inverardi, 1996] Bertolino, A. and Inverardi, P. (1996). Architecture-based software testing. In *Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops*, pages 62–64.

[Bicarregui et al., 1997] Bicarregui, J., Dick, J., Matthews, B., and Woods, E. (1997). Making the most of formal specification through animation, testing and proof. *Science of Computer Programming*, 29:53–78.

[Bidoit et al., 1999] Bidoit, M., Cengarle, M. V., and Hennicker, R. (1999). *Proof Systems for Structured Specifications and Their Refinements*, chapter 11. IFIP State-of-The-Art Reports. Springer.

[Bidoit and Hennicker, 1996] Bidoit, M. and Hennicker, R. (1996). Behavioural theories and the proof of behavioural properties. *Theoretical Computer Science*, 165(1):3–55.

[Bidoit et al., 1995] Bidoit, M., Hennicker, R., and Wirsing, M. (1995). Behavioural and abstractor implementation. *Science of Computer and Programming*, 25:149–186.

[Bougé, 1985] Bougé, L. (1985). A contribution to the theory of program testing. *Theoretical Computer Science*, 37:151–181.

[Bougé et al., 1986] Bougé, L., Choquet, N., Fribourg, L., and Gaudel, M.-C. (1986). Test set generation from algebraic specifications using logic programming. *Journal of Systems and Software*, 6(4):343–360.

[Bousquet, 1999] Bousquet, L. (1999). Feature interaction detection using testing and model-checking. In Wing, J., Woodcock, J., and Davies, J., editors, *Proceedings of FM'99*, volume 1708 of *Lecture Notes in Computer Science*, pages 622–641. Springer.

[Bowen and Hinchey, 1995] Bowen, J. P. and Hinchey, M. G. (1995). Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63.

[Bowen and Stavridou, 1993] Bowen, J. P. and Stavridou, V. (1993). Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209.

[Brinksma, 1988] Brinksma, E. (1988). A theory for the derivation of test. In Aggarwal, S. and Babnani, K., editors, *Protocol Specification, Testing and Verification VIII*. Elsevier Science Publishers.

[Brinksma, 1999] Brinksma, E. (1999). Formal methods for conformance testing: theory can be practical. In Halbwachs, N. and Peled, D., editors, *Computer Aided Verification, 11th International Conference, CAV'99*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy. Springer.

[Brinksma et al., 1995] Brinksma, E., Rensink, A., and Vogler, W. (1995). Fair testing. In Lee, I. and Smolka, S. A., editors, *CONCUR'95, Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, pages 313–327, Philadelphia. Springer.

[Burstall and Mckinna, 1992] Burstall, R. and Mckinna, J. (1992). Deliverables: an approach to program development in the calculus of constructions. Technical Report ECS-LFCS-91-133, Edinburgh University.

[Carrington and Stocks, 1994] Carrington, D. and Stocks, P. (1994). A tale of two paradigms: Formal methods and software testing. In Nicholls, J. E. and Halls, J. A., editors, *Z User's Meeting*. Springer-Verlag.

[Chen et al., 1998] Chen, H. Y., Tse, T. H., Chan, F. T., and Chen, T. Y. (1998). In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering Methodology*, 7(3):250–295.

[Clarke and Lee, 1995] Clarke, D. and Lee, I. (1995). Testing real-time constraints in a process algebraic setting. In *Proceedings of International Conference on Software Engineering - ICSE'95*, Washington USA.

[Conquet and Marty, 1999] Conquet, E. and Marty, J.-L. (1999). Formal design for automatic coding and testing: The essi/spaces project. In Wing, J., Woodcock, J., and Davies, J., editors, *Proceedings of FM'99*, volume 1708 of *Lecture Notes in Computer Science*, pages 57–75. Springer.

[Courcoubetis and Yannakakis, 1995] Courcoubetis, C. and Yannakakis, M. (1995). The complexity of probabilistic verification. *Journal of the Association for Computing Machinery*, 42(4):857–907.

[Craigen et al., 1993] Craigen, D., Gerhart, S., and Ralston, T. (1993). An international survey of industrial applications of formal methods. Technical Report NIST GCR 93/626, Atomic Energy Control Board of Canada, U.S. National Institute of Standard and Technology, and U.S. Naval Research Laboratories, National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161, USA.

[Díaz and Escrig, 1999] Díaz, L. F. L. and Escrig, D. F. (1999). Relating may ad must testing semantics for discrete timed process algebras. In Thiagarajan, P. S. and Yap, R., editors, *ASIAN'99*, volume 1742 of *Lecture Notes in Computer Science*, pages 74–86. Springer.

[Dick and Faivre, 1993] Dick, J. and Faivre, A. (1993). Automating the generation and sequencing of test cases from model-based specifications. In Wookcook, J. and Larsen, P., editors, *Formal Methods Europe - FME'93*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Springer.

[Dijkstra, 1981] Dijkstra, E. W. (1981). Why correctness must be a mathematical concern. In Boyer, R. S. and Moore, J. S., editors, *The Correctness Problem in Computer Science*. Academic Press.

[Dillon and Ramakrishna, 1996] Dillon, L. K. and Ramakrishna, Y. S. (1996). Generating oracles from your favorite temporal logic specifications. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering - SIGSOFT'96*, pages 106–117.

[Dillon and Yu, 1994] Dillon, L. K. and Yu, Q. (1994). Oracles for checking temporal properties of concurrent systems. In *ACM Sigsoft'94*, pages 140–153.

[Doche and Wiels, 2000] Doche, M. and Wiels, V. (2000). Extended institutions for testing. In Rus, T., editor, *Algebraic Methodology and Software Technology, 8th Conference, AMAST 2000*, volume 1816 of *Lecture Notes in Computer Science*, pages 514–528. Springer.

[Donat, 1997] Donat, M. R. (1997). Automating formal specification-based testing. In Bidoit, M. and Dauchet, M., editors, *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214, Lille, France. Lecture Notes in Computer Science, Springer.

[Donat, 1998] Donat, M. R. (1998). *A Discipline of Specification-Based Test Derivation*. PhD thesis, Department of Computer Science, The University of British Columbia.

[Doong and Frankl, 1994] Doong, R. and Frankl, P. G. (1994). The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130.

[Douglas and Kemmerer, 1994] Douglas, J. and Kemmerer, R. A. (1994). Aslantest: a symbolic execution tool for testing aslan formal specifications. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis, ISSTA'94*, pages 15–27, USA.

[Duke et al., 1995] Duke, R., Rose, G., and Smith, G. (1995). Object-z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533.

[Dyer, 1992] Dyer, M. (1992). *The Cleanroom Approach to Quality Software Development*. Series on Software Engineering Practice, John Wiley & Sons.

[Farrés-Casals, 1990] Farrés-Casals, J. (1990). Proving correctness w.r.t. specifications with hidden parts. In Kirchner, H. and Wechler, W., editors, *Algebraic and Logic Programming, Second International Conference*, volume 463 of *Lecture Notes in Computer Science*. Springer.

[Finney, 1996] Finney, K. (1996). Mathematical notation in formal specification: Too difficult for the masses ? *IEEE Transactions on Software Engineering*, 22(2):158–159.

[Fletcher and Sajeev, 1996] Fletcher, R. and Sajeev, A. S. M. (1996). A framework for testing object-oriented software using formal specifications. In Strohmeier, A., editor, *Reliable Software Technologies - Ada Europe 96*, volume 1088 of *Lecture Notes in Computer Science*.

[Gargantini and Heitmeyer, 1999] Gargantini, A. and Heitmeyer, C. (1999). Using model checking to generate tests from requirements specifications. In Nierstrasz, O. and Lemoine, M., editors, *ESEC/FSE'99*, volume 1687 of *Lecture Notes in Computer Science*, pages 146–162. Springer.

[Gaudel, 1995] Gaudel, M.-C. (1995). Testing can be formal, too. In Mosses, P. D., Nielsen, M., and Schwartzbach, M. I., editors, *Proceedings of Theory and Practice of Software Development - TAPSOFT'95*, volume 915 of *Lecture Notes in Computer Science*. Springer.

[Gaudel and Woodcock, 1996] Gaudel, M.-C. and Woodcock, J., editors (1996). *Third International Symposium of Formal Methods Europe, FME'96 - Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*. Springer.

[Goguen and Burstall, 1984] Goguen, J. and Burstall, R. (1984). Introducing institutions. In *Proceedings of Logic of Programming Workshop*, pages 221–256, Carnegie-Mellon.

[Goguen and Burstall, 1992] Goguen, J. A. and Burstall, R. M. (1992). Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146.

[Goodenough and Gerhart, 1975] Goodenough, J. B. and Gerhart, S. L. (1975). Towards a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173.

[Gourlay, 1983] Gourlay, J. (1983). A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, SE-9(6):686–709.

[Halbwachs and Raymond, 1999] Halbwachs, N. and Raymond, P. (1999). Validation of synchronous reactive systems: From formal verification to automatic testing. In Thiagarajan, P. S. and Yap, R., editors, *ASIAN'99*, volume 1742 of *Lecture Notes in Computer Science*, pages 1–12. Springer.

[Hall, 1996] Hall, A. (1996). Using formal method to develop an atc information system. *IEEE Software*, 13(2):66–76.

[Hall and Hierons, 1991] Hall, P. A. V. and Hierons, R. (1991). Formal methods and testing. Technical Report 91/16, The Open University Computing Department.

[Hamlet, 1994] Hamlet, D. (1994). Software quality, software process and software testing. Technical report, Center for Software Quality Research, Portland State University.

[Hennicker, 1997] Hennicker, R. (1997). *Structured Specifications with Behavioural Operators: Semantics, Proof Methods and Applications*. Habilitation thesis, Institut fur Informatik, Ludwig-Maximillians-Universitat Munchen, Munchen, Germany.

[Hoare, 1996] Hoare, C. A. R. (1996). How did software get so reliable without proof ? In Gaudel, M.-C. and Woodcock, J., editors, *Formal Methods Europe - FME'96*, volume 1051 of *Lecture Notes in Computer Science*, pages 1–17. Springer.

[Hoffman and Strooper, 1991] Hoffman, D. M. and Strooper, P. A. (1991). Automated module testing in prolog. *IEEE Transactions on Software Engineering*, 17(9):933–942.

[Holcombe, 1993] Holcombe, M. (1993). An integrated methodology for the specification, verification and testing of systems. *Software Testing, Verification and Reliability*, 3:149–163.

[Holcombe and Ipate, 1995] Holcombe, M. and Ipate, F. (1995). Almost all software testing is futile ! Technical report, FORMSOFT, Deparment of Computer Science, University of Sheffield.

[Holzmann and Smith, 1999] Holzmann, G. J. and Smith, M. H. (1999). A practical method for verifying event-driven software. In *Proceedings of International Conference on Software Engineering - ICSE'99*, pages 597–607, Los Angeles CA. ACM.

[Jagadeesan et al., 1997] Jagadeesan, L. J., Porter, A., Puchol, C., Ramming, J. C., and G.Votta, L. (1997). Specification-based testing of reactive software tools and experiments. In *Proceedings of the International Conference on Software Engineering - ICSE'97*, pages 525–535. ACM.

[Kahrs and Sannella, 1998] Kahrs, S. and Sannella, D. (1998). Reflections on the design of a specification language. In *Proceedings of International Colloquium on Fundamental Approaches to Software Engineering - ETAPS'98*, volume 1382 of *Lecture Notes in Computer Science*, pages 154–170. Springer.

[Language Design Group, 2000] Language Design Group (2000). CASL - the common algebraic specification language - summary. COFI Document (http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/).

[Laski, 1988] Laski, J. (1988). Testing in top-down program development. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, pages 72–79, Banff.

[Le Gall, 1999] Le Gall, P. (1999). *Vers une specialisation des logiques pour specifier formellement et pour tester des logiciels*. Habilitation thesis, La.M.I. Universite d'Evry, Cours Monseigneur Romero, 91025 Evry Cedex, France.

[Le Gall and Arnould, 1996] Le Gall, P. and Arnould, A. (1996). Formal specification and test: Correctness and oracle. In Haveraaen, M., Owe, O., and Dahl, O.-J., editors, *Recent Trends in Data Type Specification*, volume 1130 of *Lecture Notes in Computer Science*. Springer.

[Liu, 1999] Liu, S. (1999). Verifying consistency and validity of formal specifications by testing. In Wing, J., Woodcock, J., and Davies, J., editors, *Proceedings of FM'99*, volume 1708 of *Lecture Notes in Computer Science*, pages 896–914. Springer.

[Machado, 1998] Machado, P. D. L. (1998). Testing in stepwise formal development. In *Proceeding of the I Brazilian Workshop on Formal Methods - WMF'98*, pages 123–128, Porto Alegre, RS, Brazil. UFRGS.

[Machado, 1999] Machado, P. D. L. (1999). On oracles for interpreting test results against algebraic specifications. In Haeberer, A. M., editor, *Algebraic Methodology and Software Technology, AMAST'98*, volume 1548 of *LNCS*, pages 502–518. Springer.

[Machado, 2000a] Machado, P. D. L. (2000a). The rôle of normalisation in testing from structured algebraic specifications. In Bert, D. and Choppy, C., editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop*, volume 1827 of *Lecture Notes in Computer Science*, pages 459–476. Springer.

[Machado, 2000b] Machado, P. D. L. (2000b). Testing from structured algebraic specifications. In Rus, T., editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST'00*, volume 1816 of *Lecture Notes in Computer Science*, pages 529–544, Iowa, USA. Springer.

[Machado and Meira, 1995] Machado, P. D. L. and Meira, S. L. (1995). On the use of formal specifications in the design and simulation of artificial neural networks. In Bowen, J. P. and Hinchey, M. G., editors, *ZUM'95: The Z Formal Specification Notation*, volume 967 of *Lecture Notes in Computer Science*, pages 63–82. Springer-Verlag.

[Mandrioli et al., 1995] Mandrioli, D., Morasca, S., and Morzenti, A. (1995). Generating test cases for real-time systems from logic specifications. *ACM Transactions on Computer Systems*, 13(4):365–398.

[Manna and Waldinger, 1981] Manna, Z. and Waldinger, R. (1981). Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1:5–48.

[McDonald et al., 1997] McDonald, J., Murray, L., and Strooper, P. (1997). Translating object-z specifications to object-oriented test oracles. Technical Report 97-30, Software Verification Research Centre, The University of Queensland, Australia. Version 2, March 1998.

[Mikk, 1995] Mikk, E. (1995). Compilation of z specifications in c for automatic test result evaluation. In Bowen, J. P. and Hinchey, M. G., editors, *Proceedings of the Ninth Annual Z User Meeting*, volume 967 of *Lecture Notes in Computer Science*, pages 167–180. Springer.

[Milner et al., 1997] Milner, R., Tofte, M., Harper, R., and MacQueen, D. (1997). *The Definition of Standard ML (Revised)*. The MIT Press.

[Morasca et al., 1996] Morasca, S., Morzenti, A., and SanPietro, P. (1996). Generating functional test cases in-the-large for time-critical systems from logic-based specifications. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis - ISSTA'96*, pages 39–52.

[Müllerburg et al., 1995] Müllerburg, M., Holenderski, L., Maffeis, O., Merceron, A., and Morley, M. (1995). Systematic testing and formal verification to validate reactive programs. *Software Quality Journal*, 4(4).

[Murray et al., 1998] Murray, L., Carrington, D., MacColl, I., McDonald, J., and Strooper, P. (1998). Formal derivation of finite state machines for class testing. Technical Report 98-03, Software Verification Research Centre, The University of Queensland, Queensland 4072.

[O'Malley et al., 1996] O'Malley, T. O., Richardson, D. J., and Dillon, L. K. (1996). Efficient specification-based oracles for critical systems. In *Proceedings of California Software Symposium*.

[Ould and Unwin, 1986] Ould, M. A. and Unwin, C., editors (1986). *Testing in Software Development*. Cambridge University Press. by Members of the British Computer Society Working Group on Testing.

[Parissis and Ouabdesselam, 1996] Parissis, I. and Ouabdesselam, F. (1996). Specification-based testing of synchronous software. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 127–134.

[Paulson, 1996] Paulson, L. C. (1996). *ML for the working programmer*. Cambridge University Press, 2 edition.

[Péraire, 1998] Péraire, C. (1998). *Formal Testing of Object-Oriented Software: from the Method to the Tool*. PhD thesis, École Polytechnique Fédérale de Lausanne.

[Péraire et al., 1998] Péraire, C., Barbey, S., and Buchs, D. (1998). Test selection for object-oriented software based on formal specifications. In *IFIP Working Conference on Programming Concepts and Methods (PROCOMET'98)*, pages 385–403, Shelter Island, New York, USA. Chapman & Hall.

[Pressman, 1994] Pressman, R. (1994). *Software Engineering*. McGrawHill.

[Rice and Seidman, 1998] Rice, M. D. and Seidman, S. B. (1998). An approach to architectural analysis and testing. In *Proceedings of the Third International Workshop on Software Architecture*, pages 121–123.

[Richardson et al., 1989] Richardson, D., O'Malley, T., and Moore, C. T. (1989). Approaches to specification based testing. *ACM Sigsoft 89: Third Symposium on Software Testing, Analysis and Verification*.

[Richardson et al., 1992] Richardson, D. J., Aha, S. L., and O'Malley, T. O. (1992). Specification-based test oracles for reactive systems. In *Proceedings of the Fourteenth International Conference on Software Engineering - ICSE'92*, pages 105–118, Melbourne, Australia.

[Richardson and Wolf, 1996] Richardson, D. J. and Wolf, A. L. (1996). Software testing at the architectural level. In *Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops*, pages 121–123.

[Saiedian, 1996] Saiedian, H. (1996). An invitation to formal methods. *IEEE Computer*, pages 16–30.

[Sankar et al., 1993] Sankar, S., Goyal, A., and Sikchi, P. (1993). Software testing using algebraic specification based test oracles. Technical Report CSL-TR-93-566, Computer System Laboratory, Stanford University, Stanford, California 94305-4055.

[Sannella et al., 1992] Sannella, D., Sokolowski, S., and Tarlecki, A. (1992). Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Informatica*, 29(8):689–736.

[Sannella and Tarlecki, 1997] Sannella, D. and Tarlecki, A. (1997). Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269.

[Sannella and Tarlecki, 1987] Sannella, D. T. and Tarlecki, A. (1987). On observational equivalence and algebraic specification. *Journal of Computer and System Sciences*, 34(2/3):150–178.

[Schneider, 1999] Schneider, S. (1999). Abstraction and testing. In Wing, J., Woodcock, J., and Davies, J., editors, *Proceedings of FM'99*, volume 1708 of *Lecture Notes in Computer Science*, pages 738–757. Springer.

[Semantics Group, 1999] Semantics Group (1999). CASL - the common algebraic specification language - semantics. CoFI Document (http://www.brics.dk/Projects/CoFI/Documents/CASL/Semantics/).

[Sommerville, 1995] Sommerville, I. (1995). *Software Engineering*. Addison-Wesley, fifth edition.

[Stepney, 1995] Stepney, S. (1995). Testing as abstraction. In Bowen, J. P. and Hinchey, M. G., editors, *Lecture Notes in Computer Science, ZUM'95: The Z Formal Specification Notation*, volume 967, pages 137–151. Springer-Verlag.

[Stocks and Carrington, 1996] Stocks, P. and Carrington, D. (1996). A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793.

[Tabourier et al., 1999] Tabourier, M., Cavalli, A., and Ionescu, M. (1999). A gsm-map protocol experiment using passive testing. In Wing, J., Woodcock, J., and Davies, J., editors, *Proceedings of FM'99*, volume 1708 of *Lecture Notes in Computer Science*, pages 915–934. Springer.

[Wirsing, 1990] Wirsing, M. (1990). Algebraic specification. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, volume B, chapter 13, pages 675–788. Elsevier Science Publishers.

[Zhu et al., 1997] Zhu, H., Hall, P. A. V., and May, J. H. R. (1997). Software unit test coverage and adequacy. *ACM Comput. Survey*, 29(4):366–427.

Index