Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 130 (2005) 235–261
www.elsevier.com/locate/entcs

Testing from Structured Algebraic Specifications: The Veritas Case Study[⋆]

Patrícia D. L. Machado[1,2,3], Elthon A. S. Oliveira[1,4], Paulo E. S. Barbosa[1,5], Cássio L. Rodrigues[1,6]

Departamento de Sistemas e Computação
Universidade Federal de Campina Grande
Campina Grande, Brazil

Abstract

The use of algebraic specification-based testing to validate applications implemented in SML is discussed, particularly the VERITAS model-checker. Test case, oracle and data are generated from structured specifications in CASL with test oracles being responsible for driving and interpreting the results of tests according to fundamental research in the area. The objective of this work is twofold – to test conformance of the VERITAS model checker with respect to a structured algebraic specification, and to contribute to further development in the area of specification-based testing by illustrating its application, focusing on theoretical problems and solutions anticipated.

Keywords: Specification-based testing, structured specification, algebraic specification, oracle problem, test harness.

⋆ Extended version of the paper "Algebraic Specification-Based Testing: The Veritas Case Study" presented at the SBMF 2004.

1 Supported by CNPq - Brazilian Research Council, Process 552190/2002-0 and 477808/2003-4

2 Supported by CNPq/FAPESQ - Fundação de Apoio a Pesquisa do Estado da Paraíba, Project 060/03

3 Email: `patricia@dsc.ufcg.edu.br`

4 Email: `elthon@dsc.ufcg.edu.br`

5 Email: `paulo@dsc.ufcg.edu.br`

6 Email: `caleo@dsc.ufcg.edu.br`

1 Introduction

Specification-based testing (SBT) is concerned with deriving test suites from formal specifications of programs. More recently, several works in this area have been developed [3,13,2], promoting the combined use of formal methods and testing to produce high integrity systems in a cost-effective way [7,4]. In the algebraic field, SBT consists in checking whether specification axioms are satisfied by an implementation under test (IUT). From a selected test case (usually an axiom), tests are run to exercise referred operations and an oracle evaluate the output criteria according to the results produced by the tests [15,21]. In other words, oracles check satisfaction of specification axioms, for a finite test data set, by the IUT.

In order to establish testing as an effective verification technique, it is essential to develop well-founded methods and strategies that support automation of testing activities [19]. A great effort is still needed to have testing as a standard activity in formal frameworks. Accurate interpretation of test results regarding correctness and how to properly select finite test sets along with automation and technology transfer are crucial points. Exploratory attempts to apply formal SBT in industrial settings can already be found [2].

We present a case study of SBT to validate the Veritas tool [27,28] – a model checker for an object-oriented Petri nets modelling language called RPOO [16]. Veritas uses CTL temporal logic [11] for properties specification. The tool exploits the object-oriented view of RPOO models so that we do not need to deal with Petri nets syntax for specifying atomic propositions. Veritas is implemented in the SML language.

Our main contribution is to exemplify the use of algebraic SBT, focusing on proposed theoretical solutions and results presented in the literature, in a real case study, uncovering advantages and limitations. Moreover, we aim at contributing to efforts on reducing the gap between theory and practice on SBT. This kind of study is crucial to make technology transfer possible. Furthermore, and not less important, we check conformance of the implementation of the Veritas model checker with respect to a structured algebraic specification in Casl[6]. The Veritas case study is particularly suitable for SBT as both code and data manipulated are complex enough to make full formal verification unfeasible.

The paper is structured as follows. Section 2 presents basic terminology on algebraic specifications, testing satisfaction and the SBT approach to be followed. Section 3 introduces the case study and presents its formal specification in Casl. Section 4 presents the methodology followed to conduct the case study. Section 5 discusses the results obtained and lessons learned. Finally, Section 6 presents concluding remarks along with pointers for further work.

2 Background

This section presents basic terminology on algebraic specifications, testing satisfaction and the SBT approach followed in this paper.

2.1 Algebraic Specifications

As a usual assumption, implementations are modelled as algebras. Also, a specification declares a set of symbols – the signature – and contains axioms giving required properties of these symbols. Let $\Sigma = (S, F, Obs)$ be a *signature*[7] with $sorts(\Sigma) = S$, $opns(\Sigma) = F$ and $Obs \subseteq S$ – a set of observable sorts[8]. Let $T_\Sigma(X)$ be the *term algebra* (values are terms built from Σ and X), where X is an S-indexed set of countably infinite sets of variables. For any two terms t and t' of the same sort, $t = t'$ is an *equation*; *first-order formulas* are built from equations, logical connectives ($\neg$,$\wedge$,$\vee$,$\Rightarrow$,$\Leftrightarrow$) and quantifiers ($\forall$,$\exists$). Axioms in specifications are formulas without free variables, called *sentences*.

A Σ-*algebra* A consists of a S-sorted set $|A|$, the *carrier sets*, and for each $f : s_1 \times \ldots \times s_n \to s$ in Σ, a function $f_A : |A|_{s_1} \times \ldots \times |A|_{s_n} \to |A|_s$. We restrict to algebras with non-empty carriers. For any Σ-algebra A and valuation $\alpha : X \to |A|$, there is a unique homomorphism $\alpha^\# : T_\Sigma(X) \to A$ which extends α (translation function). The *value* of $t \in |T_\Sigma(X)|_s$ in A under α is then $\alpha^\#(t) \in |A|_s$. If $t \in T_\Sigma$, i.e., t is a *ground term*, the value of t in A is ${}^\#(t)$, where ${}^\# : T_\Sigma \to A$ is the unique homomorphism.

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.

2.2 Behavioural and Approximate Equality

The equality problem – an instance of the oracle problem – is the question of how equality on non-observable sorts is defined, where a non-observable sort is one that is not identified with any particular concrete representation or standard datatype. Therefore, it is not appropriate to assume that equality on values of this sort is the usual set-theoretical one. Equality on values of a Σ-algebra A can be interpreted by an appropriate *behavioural equality*. This is a

[7] Some specification languages, including CASL, permit signatures to include *predicates*. W.l.o.g. we will regard these as operations yielding a boolean result.

[8] Observable sorts are usually identified with predefined ones in programming languages.

partial congruence $\approx_A = (\approx_{A,s})_{s\in S}$ (one relation for each sort $s \in S$) of partial equivalence relations – symmetric and transitive relations – which are compatible with Σ [9]. The domain of definition of $\approx_A$ is $Dom(\approx_A) = \{a \mid a \approx_A a\}$. When A is obvious, $\approx$ is used to denote $\approx_A$. 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 one example of a behavioural equality, where related elements are those that cannot be distinguished by observable computations. [10]

Behavioural equality can be difficult to test. Consider e.g. observational equality on non-observable sorts which is defined in terms of a set of contexts that is usually infinite. One approach involves the use of *approximate equalities* [20] which are binary relations on values of the algebra. When compared to a behavioural equality, an approximate equality is *sound* if all values (in $Dom(\approx_A)$) that it identifies are indeed equal, or *complete* if all equal values are identified (*sound* $\subseteq$ *behavioural* $\subseteq$ *complete*). A *contextual equality* $\sim_{C,A}$ is defined from a subset C (usually finite) of the observable computations rather than the set of all observable computations. Any contextual equality is complete with respect to observational equality, although it is not necessarily a partial congruence. The set-theoretical equality is sound – in programming terms, this is equality on the underlying data representation.

2.3 Testing Satisfaction

We use a notion of satisfaction of axioms by algebras named testing satisfaction relation with equality interpreted by approximate equalities and quantifiers ranging over test sets – ground terms derived from the specification [20].

Definition 2.1 (Testing Satisfaction) *Let Σ be a signature, $T \subseteq T_\Sigma$ be a test set and $\sim, \doteq$ be two approximate equalities on a Σ-algebra A. Let $\alpha : X \to Dom(\approx_A)$ be a valuation. The* testing satisfaction relation *denoted by $\models^T$ is defined as follows.*

(i) $A, \alpha, \sim, \doteq\ \models^T t = t'$ *iff* $\alpha^\#(t) \sim_A \alpha^\#(t')$*;*

(ii) $A, \alpha, \sim, \doteq\ \models^T \neg\psi$ *iff* $A, \alpha, \doteq, \sim\ \models^T \psi$ *does not hold;*

(iii) $A, \alpha, \sim, \doteq\ \models^T \psi_1 \wedge \psi_2$ *iff both* $A, \alpha, \sim, \doteq\ \models^T \psi_1$ *and* $A, \alpha, \sim, \doteq\ \models^T \psi_2$ *hold;*

[9] $\approx_A$ is compatible with Σ iff $\forall f : s_1 \dots s_n \longrightarrow s \in F$, $\forall a_i, b_i \in A_{s_i}$, if $a_i \approx_{A,s_i} b_i$ for all $1 \le i \le n$, then $f_A(a_1, \dots, a_n) \approx_{A,s} f_A(b_1, \dots, b_n)$

[10] Let C_{Obs} be the set of all 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*, $a \approx_{Obs,A,s} b$, iff $a, b \in {}^\#(T_\Sigma)$ 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 reachable subalgebra of A and $\alpha_a, \alpha_b : X \cup \{z_s\} \to {}^\#(T_\Sigma)$ are the extensions of α defined by $\alpha_a(z_s) = a$ and $\alpha_b(z_s) = b$.

(iv) $A, \alpha, \sim, \doteq \models^T \forall x{:}s \cdot \psi$ *iff* $A, \alpha[x \mapsto v], \sim, \doteq \models^T \psi$ *holds for all* $v \in {}^{\#}(T)_s$*;*

where $\alpha[x \mapsto v]$ *denotes the valuation* α *superseded at* x *by* v*. Satisfaction of formulae involving* $\vee$*,* $\Rightarrow$*,* $\Leftrightarrow$*,* $\exists$ *is defined using the usual definitions of these in terms of* $\neg$*,* $\wedge$*,* $\forall$*. In this relation,* $\sim$ *is always applied in positive contexts and* $\doteq$ *is always applied in negative contexts*[11]*. Note that the approximate equalities are reversed when negation is encountered.*

The following theorem relates testing satisfaction to usual behavioural satisfaction ($\models$), where equality is interpreted as behavioural equality ($\approx$) and quantification is over all of $Dom(\approx)$. Note that behavioural satisfaction coincides with our notion of correctness.

Theorem 2.2 **([20])** *If* $\sim$ *is complete,* $\doteq$ *is sound, and* ψ *has only positive occurrences of* $\forall$ *and negative occurrences of* $\exists$*, then* $A, \alpha, \approx \models \psi$ *implies* $A, \alpha, \sim, \doteq \models^T \psi$*.* $\square$

The restriction to positive $\forall$ and negative $\exists$ is not a problem in practice, since it is satisfied by most common specification idioms.

Theorem 2.2 implies that incorrect programs can be accepted by the testing relation, but failure in testing means incorrectness. On the other hand, the dual of this theorem, which covers a more rare situation, implies that correct programs can be rejected, but success in testing means correctness [20]. Assumptions on quantifiers can be dropped if T is unbiased[12]. In practice, this result show that, under the conditions stated, approximate equalities can be applied together with finite test sets to detect incorrectness without leading to rejection of correct implementations. Encapsulation and information hiding can make it impossible to defined very precise equalities without referring to internal information. This information may not be available due to reusability and context independence design goals – a common practice of software components development methods [8]. Furthermore, even if we have all observations at hand, they may require arguments of other sets of values that may be infinite or too big. In this case, it will be impossible or impractical to run the real equality that considers all possible values of these sets.

[11] A context is *positive* if it is formed by an even number of applications of negation (e.g. ϕ is in a positive context in both $\phi \wedge \psi$ and $\neg\neg\phi$). Otherwise, the context is *negative*. Note that ϕ is in a negative context and ψ is in a positive context in $\phi \Rightarrow \psi$ since it is equivalent to $\neg\phi \vee \psi$. A formula or symbol *occurs positively* (resp. *negatively*) in ϕ if it occurs in a positive (resp. negative) context within ϕ.

[12] T is unbiased iff $A, \alpha, \sim, \doteq \models^{T_\Sigma} \psi$ implies $A, \alpha, \sim, \doteq \models^T \psi$. T is valid iff $A, \alpha, \sim, \doteq \models^T \psi$ implies $A, \alpha, \sim, \doteq \models^{T_\Sigma} \psi$, where T_Σ is the exhaustive test set.

2.4 The Grey-Box Approach

The testing satisfaction relation can be used as a basis to implement approximate oracles for interpreting the results of tests. The grey-box approach aims at producing approximate oracles according to Theorem 2.2 and its variants [20,21,22]. The idea is to use white-box techniques to produce a sound equality and black-box techniques to produce a complete equality from a finite subset of observable computations. A combination of white-box and black-box techniques to sort out the equality problem can also be found in [13,9]. The reason for defining two equalities is that each one can be successfully applied in contexts where the other might not be.

Equalities defined from a subset of the set of all observable contexts – contextual equalities – are always complete w.r.t. observational equality. The equality induced by these contexts either coincides with the observational equality or is a complete approximate equality. Structural equalities based on the equality of the values of the concrete representation of a sort, 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 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 grey-box approach can be applied with the following purposes: (i) Attempt to detect incorrectness without rejecting correct implementations, by 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 implementations by 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 implementation under test (IUT) is incorrect. However, it is easy to check that the converse does not hold: if testing is successful we cannot conclude that the IUT is correct. On the other hand, whenever testing is successful in (ii) we can conclude that the IUT is correct, but once again the converse does not hold, i.e., if testing is not successful, we cannot conclude the IUT is incorrect. Furthermore, if both approaches can be applied (T is both valid and unbiased) and the test fails in approach (ii) and, thereafter, in (i), the IUT is incorrect while if it succeeds in approach(i) and, thereafter, in (ii), then the IUT is correct [20].

2.5 Testing from Structured Specifications

Even if the signature of the IUT matches the signature of the specification, the structure of the IUT does not necessarily reflect the structure of the specification. On the other hand, when testing from a structured specification, it is necessary to think of its structure. This is due to the fact that the semantics of specifications is given in a compositional way [5], i.e., the signature and class of models of a specification are determined according to the result of applying specification-building operations to its constituent specifications. In other words, the structure of the specification must be considered in order to make sense of axioms. Accordingly, among obstacles that can be encountered when testing from structured specifications are [22]:

(i) 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.

(ii) 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.

(iii) It is reasonable to think that particular test sets for a given sort should be defined separately for some signatures, groups of axioms or individual axioms. In other words, it may be beneficial to handle the quantifier problem differently according to the signature/axiom 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 [21].

Definition 2.3 (Structured Specifications with Test Sets) *The syntax and semantics of structured specifications are inductively defined as follows. The semantics is given in terms of signature and classes of models.*

(i) $SP = \langle \Sigma, \Psi \rangle$ *with* $\Psi \subseteq \{(\psi, T) \mid \psi \in Sen(\Sigma)$ and $T \subseteq T_\Sigma\}$ *is defined as follows.*

- $Sig(SP) \overset{def}{=} \Sigma$
- $Mod_{\approx}(SP) \overset{def}{=} \{A \in Alg(\Sigma) \mid \bigwedge_{(\psi,T)\in\Psi} A, \approx_A \models \psi\}$

- $ChMod_{\sim,\doteqdot}(SP) \overset{def}{=} \{A \in Alg(\Sigma) \mid \bigwedge_{(\psi,T)\in\Psi} A, \sim_A, \doteqdot_A \models^T \psi\}$

(ii) *$SP = SP_1 \cup SP_2$, where SP_1 and SP_2 are structured specifications, with $Sig(SP_1) = Sig(SP_2)$.*

- $Sig(SP) \overset{def}{=} Sig(SP_1) = Sig(SP_2)$
- $Mod_{\approx}(SP) \overset{def}{=} Mod_{\approx}(SP_1) \cap Mod_{\approx}(SP_2)$
- $ChMod_{\sim,\doteqdot}(SP) \overset{def}{=} ChMod_{\sim,\doteqdot}(SP_1) \cap ChMod_{\sim,\doteqdot}(SP_2)$

(iii) *$SP =$ translate SP' with σ, where $\sigma : \Sigma' \to \Sigma$ and $Sig(SP') = \Sigma'$.*

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

(iv) *$SP =$ hide sorts S' opns F' in 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.*

- $Sig(SP) \overset{def}{=} Sig(SP') - \langle S', F'\rangle$
- $Mod_{\approx}(SP) \overset{def}{=} \{A'|_\Sigma \mid A' \in Mod_{\approx}(SP')\}$
- $ChMod_{\sim,\doteqdot}(SP) \overset{def}{=} \{A'|_\Sigma \mid A' \in ChMod_{\sim,\doteqdot}(SP')\}$

where $Mod_{\approx}(SP)$ is the class of "real" models *of SP w.r.t. the family of Σ-behavioural equalities $\approx = (\approx_\Sigma)_{\Sigma\in Sign}$ and $ChMod_{\sim,\doteqdot}(SP)$ is the class of* "checkable" models *of SP by testing w.r.t. the families of Σ-approximate equalities $\sim = (\sim_\Sigma)_{\Sigma\in Sign}$ and $\doteqdot = (\doteqdot_\Sigma)_{\Sigma\in Sign}$. Note that test sets are defined at specification level and associated with axioms.*

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 [31,18], like *enrichment* (**then** in CASL) and *arbitrary union* or sum of specifications (**and** in CASL). Instantiation of generic specifications can be defined in terms of *union* and *translate* in the usual way. For example, see [29].

Theorem 2.4 below is a generalisation of Theorem 2.2 for structured specifications. Under certain conditions incorrect programs can be accept by testing, i.e., 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 2.4 ([21]) *If $\sim$ is complete, $\doteqdot$ 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* $A \in ChMod_{\sim,\simeq}(SP)$.

It is important to consider the structure of the specification during test planning, since this can greatly influence the way test cases and test data are generated and test harness is constructed.

3 The Veritas Model Checker

In this section, we present an overview of the Veritas model checker and a Casl specification of its property evaluation module. The specification is used as a basis to generate test cases to validate the Veritas implementation.

3.1 Overview

Model checking is an automatic technique for verifying finite state concurrent systems. It provides the means for checking that a model of a design satisfies a given specification [12]. To verify an RPOO model using Veritas, we need to provide the state space of the model and a specification of the property we want to check. The verification process and the tool architecture is shown in Figure 1.

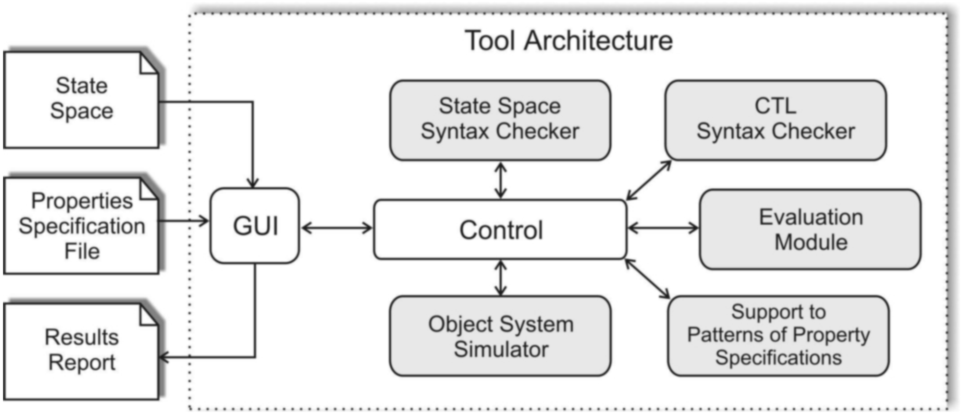

Fig. 1. The verification process and the tool architecture.

The *Evaluation module* is the main part of the tool. It is responsible for verifying the described properties in the specification against the provided state space. In general, if a universally quantified formula is evaluated as false, the evaluation module shows a *counter-example* trace which proves that the property is not true in the system. In a similar way, when an existentially quantified formula is true, the module shows a *witness* trace which proves that the property is satisfied in the system. The implemented algorithms are based on the forward transition relation; and they perform the evaluation process and the trace construction at the same time.

Veritas does not construct the state space of the system; this structure must be provided as a parameter of the verification process. This is due to an architectural feature that promotes low coupling between the model checker and the tool for state space generation. Consequently, the state space explosion problem [30] can be treated in an independent way of the model checking problem. The tools integration is achieved by means of a standard format for state space representation. For this reason, there is a module in the tool to verify whether the provided state space complies with such standard format. This module is named *State Space Syntax Checker*. In an analogous way, we check whether the properties to be verified also comply with the CTL syntax. However, this is done by a specific module, called *CTL Syntax Checker*. In order to increase the automation degree of the specification task, there is a module in the tool that implements patterns of property specification for finite-state verification, proposed by Dwyer [14]. This module is named *Support to Patterns of property Specifications*.

Finally, the results produced by the evaluation module can be viewed as a textual report or as a graphical representation. In the former mode, the report contains the truth-value of the property, the possible trace proving such value, and the CPU time spent in the evaluation process. In the latter, we get a visualisation of the traces, which allows us to analyse them step-by-step in a graphical mode. This is done by the *Object System Simulator* module.

3.2 Algebraic Specification

This section presents a specification in Casl of part of the evaluation module of Veritas. For the sake of simplicity, the presentation focus on the checking result, excluding traces associated with the result that are also returned by the tool. The specification is divided into three parts: Kripke, Path and CTLOperators. We expect the notation to be mostly self-explanatory. Note that, for the purposes of Section 5, axioms are labelled according to a numbering of test cases.

The Kripke specification (Figure 2) defines the state space structure used by Veritas – a finite state machine that represents all possible executions of a system. The main data involved are kripke structures (*Kripke* sort), states (*State* sort) and state identifiers (*Nat* sort). The specification imports the basic specification of natural numbers, Nat, and the generic specification of set, Set, instantiated as Set[Nat]. Note that axioms are implicitly universally quantified by all the declared variables. Any state in a kripke structure

spec KRIPKE = SET [NAT **fit** *Elem* $\mapsto$ *Nat*] **then**
  sorts *Kripke*, *State*;
  ops $\mathit{stateId} : \mathit{State} \rightarrow \mathit{Nat}$;
    $\mathit{getState} : \mathit{Nat} \times \mathit{Kripke} \rightarrow ?\ \mathit{State}$;
    $\mathit{getSuc} : \mathit{Nat} \times \mathit{Kripke} \rightarrow ?\ \mathit{Set}[\mathit{Nat}]$;
    $\mathit{getPred} : \mathit{Nat} \times \mathit{Kripke} \rightarrow ?\ \mathit{Set}[\mathit{Nat}]$;
    $\mathit{allStates} : \mathit{Kripke} \rightarrow \mathit{Set}[\mathit{Nat}]$;
  pred $\mathit{isNext} : \mathit{Nat} \times \mathit{Nat} \times \mathit{Kripke}$;
  vars $s1, s2 : \mathit{Nat};\ k : \mathit{Kripke};\ t1, t2 : \mathit{State}$
- $\mathit{isNonEmpty}(\mathit{getSuc}(s1, k))$ **if** $s1\ \mathit{eps}\ \mathit{allStates}(k)$ %(0)%
- $\mathit{def}\ \mathit{getState}\ (s1, k) \Leftrightarrow s1\ \mathit{eps}\ \mathit{allStates}(k)$ %(1,2)%
- $\mathit{def}\ \mathit{getSuc}(s1, k) \Leftrightarrow \mathit{def}\ \mathit{getState}(s1, k)$ %(3,4)%
- $\mathit{def}\ \mathit{getPred}(s1, k) \Leftrightarrow \mathit{def}\ \mathit{getState}(s1, k)$ %(5,6)%
- $(s2\ \mathit{eps}\ \mathit{getSuc}(s1, k) \Leftrightarrow \mathit{isNext}(s2, s1, k))$ **if** $s1\ \mathit{eps}\ \mathit{allStates}(k)$ %(7,8)%
- $(s1\ \mathit{eps}\ \mathit{getPred}(s2, k) \Leftrightarrow \mathit{isNext}(s2, s1, k))$
  if $s2\ \mathit{eps}\ \mathit{allStates}(k)$ %(9,10)%
- $\mathit{stateId}(\mathit{getState}(s1, k)) = s1$ **if** $s1\ \mathit{eps}\ \mathit{allStates}(k)$ %(11)%
- $(\mathit{getState}(s1, k) = \mathit{getState}(s2, k) \Rightarrow s1 = s2)$
  if $s1\ \mathit{eps}\ \mathit{allStates}(k) \wedge s2\ \mathit{eps}\ \mathit{allStates}(k)$ %(12)%
- $(\mathit{stateId}(t1) = \mathit{stateId}(t2) \Rightarrow t1 = t2)$
  if $\mathit{stateId}(t1)\ \mathit{eps}\ \mathit{allStates}(k) \wedge \mathit{stateId}(t2)\ \mathit{eps}\ \mathit{allStates}(k)$ %(13)%
- $\mathit{getSuc}(s1, k)\ \mathit{isSubsetOf}\ \mathit{allStates}(k)$ **if** $s1\ \mathit{eps}\ \mathit{allStates}(k)$ %(14)%
- $\mathit{getPred}(s1, k)\ \mathit{isSubsetOf}\ \mathit{allStates}(k)$ **if** $s1\ \mathit{eps}\ \mathit{allStates}(k)$ %(15)%

end

Fig. 2. The KRIPKE Specification.

must have a successor (Axiom (0)). Given a state identifier, *getState*, *getSuc* and *getPred* returns the state, its successor and predecessor identifiers, respectively. These operations are only defined for state identifiers that belongs to the set of all states of a kripke structure – *allStates* operation (Axioms (1,2), (3,4), (5,6), (14), (15)). The transition relation is expressed by the *isNext* predicate (Axioms (7,8), (9,10)). Finally, state identifiers are unique and no state can have two different identifiers (Axioms (11), (12) and (13)).

The PATH specification, presented in Figure 3, defines paths. A path in a kripke structure k is an infinite sequence of states $s_0, s_1, s_2, \ldots$ s.t. s_0 is the initial state and s_{i+1} is a successor of s_i in k for all $i \geq 0$. The specification imports the KRIPKE specification, the LIST specification instantiated as LIST[NAT] and the SET specification instantiated as SET[PATH]. The *paths* operation returns all valid paths with a given state s as the initial state (Axiom (2,3)). A path is valid if and only if the transition rules of the kripke structure are observed by the states ordering in the path (Axiom (4,5)). Paths can be constructed from a state list, but the *constructPath* operation is only defined for valid paths (Axiom (0)) and the state list is preserved (Axiom (1)).

spec SortPath = **sort** *Path* **end**

spec Path = Kripke **and** List [Nat **fit** *Elem* $\mapsto$ *Nat*]
  and Set[SortPath **fit** *Elem* $\mapsto$ *Path*] **then**
  ops $constructPath : List[Nat] \times Kripke \to ?\ Path;$
    $allPathStates : Path \times Kripke \to List[Nat];$
    $paths : Nat \times Kripke \to Set[Path];$
  pred $isValid : Path \times Kripke;$
  vars $p : Path;\ k : Kripke;\ l : List[Nat];$

- $def\ constructPath(l, k) \Rightarrow isValid(constructPath(l, k), k)$ %(0)%
- $def\ constructPath(l, k) \Rightarrow allPathStates(constructPath(l, k), k) = l$ %(1)%
- $(\exists s : Nat \cdot (p\ eps\ paths(s, k)) \Leftrightarrow isValid(p, k)$ %(2, 3)%
- $isValid(p, k) \Leftrightarrow \forall j : Nat \cdot (j < \#(allPathStates(p, k)) -! 1 \Rightarrow$
  $(allPathStates(p, k)!j\ eps\ allStates(k) \wedge$
  $allPathStates(p, k)!(j + 1)\ eps\ allStates(k) \wedge$
  $isNext(allPathStates(p, k)!(j + 1), allPathStates(p, k)!(j), k))))$ %(4, 5)%

end

Fig. 3. The Path Specification.

The CTLOperators specification (Figures 4 and 5) defines the syntax and semantics of CTL (Computation Tree Logic) formulas. The specification imports the Path specification and the String specification. The main additional data involved are CTL formulas (*CTLF* sort) and atomic propositions (*PROPEVAL* sort). A CTL formula can be constructed from an atomic proposition using the *AP* operation. The *eval* operation represents atomic proposition evaluation for a given kripke structure and a state. *TT* and *FF* are trivial CTL formulas. Given two CTL formulas ϕ and ψ, *NOTCTL*(ϕ), *ANDCTL*(ϕ, ψ), *ORCTL*(ϕ, ψ), *EU*(ϕ,ψ), *AU*(ϕ,ψ), *EF*(ϕ), *AF*(ϕ), *EX*(ϕ), *AX*(ϕ), *EG*(ϕ), *AG*(ϕ) are also CTL formulas.

CTLOperators axioms (Figure 5) define the semantics of a formula satisfaction (*check* operation) for a given kripke structure k and initial state s [11]. Note that the semantics of *AX*, *EF*, *AG*, *AF* and *AU* is given in terms of *EX*, *EU* and *EG* (Axioms (20,21), (14,15), (24,25), (16,17), (12,13)). Actually, Veritas implements only the latter ones and normalises the former ones using the same equivalence laws used in the CTLOperators specification. Note that $EX\phi$ means that there exists a successor state of the current one so that ϕ is valid in this state (Axiom (18,19); $E[\phi U \psi]$ means that there exists a path from the current state so that ϕ must be valid in all states of the path until a state is reached with ψ valid (Axiom (10,11)); and $EG(\phi)$ means that there exists a path from the current state with ϕ always valid (Axiom (22,23)). The

```
spec CTLOperators = Path and String then
  sorts CTLF, PROPEVAL;
  ops TT : CTLF;
      FF : CTLF;
      AP : String × PROPEVAL → CTLF;
      NOTCTL : CTLF → CTLF;
      ANDCTL : CTLF × CTLF → CTLF;
      ORCTL : CTLF × CTLF → CTLF;
      EU : CTLF × CTLF → CTLF;
      AU : CTLF × CTLF → CTLF;
      EF : CTLF → CTLF;
      AF : CTLF → CTLF;
      EX : CTLF → CTLF;
      AX : CTLF → CTLF;
      EG : CTLF → CTLF;
      AG : CTLF → CTLF;
  preds check : Kripke × Nat × CTLF; eval : Kripke × Nat × PROPEVAL;
  ...
end
```

Fig. 4. The CTLOperators Specification (sorts, operations, predicate signature).

ANDCTL, *ORCTL* and *NOTCTL* are specified as usual ((4,5), (6,7), (8,9)).

4 The Case Study Methodology

This work is aimed at producing automated functional tests, generated from a formal specification, that can be run to validate the implementation of Veritas. Also, to reduce the gap between model checking theory and a concrete implementation of it by having automated test suites that can evolve with the formal specification and be effectively applied whenever the implementation changes. This is very important for complex systems such as Veritas that are continuously evolving to meet performance requirements.

Veritas has been implemented in the SML language according to algorithms of CTL formula checking proposed in [27]. Both algorithms and implementation were validated, prior to the experiment presented in this paper, based on static analysis and ad-hoc/manual testing. However, the complexity of the implementation of such applications naturally makes static analysis hard or even impossible to be fully performed.

The validation process used in the case study presented in this paper has four main activities: 1) Test planning; 2) Formal specification in Casl; 3) Test harness construction; 4) Test suites generation, execution and analysis. These

vars $k : \mathit{Kripke}$; $s : \mathit{Nat}$; $i : \mathit{String}$; $p : \mathit{PROPEVAL}$; $f1, f2 : \mathit{CTLF}$

- $\mathit{notcheck}(k, s, FF)$ %(0)%
- $\mathit{check}(k, s, TT)$ %(1)%
- $\mathit{check}(k, s, AP(i, p)) \Leftrightarrow \mathit{eval}(k, s, p)$ %(2, 3)%
- $\mathit{check}(k, s, \mathit{NOTCTL}(f1)) \Leftrightarrow \neg \mathit{check}(k, s, f1)$ %(4, 5)%
- $\mathit{check}(k, s, \mathit{ANDCTL}(f1, f2)) \Leftrightarrow (\mathit{check}(k, s, f1) \wedge \mathit{check}(k, s, f2))$ %(6, 7)%
- $\mathit{check}(k, s, \mathit{ORCTL}(f1, f2)) \Leftrightarrow (\mathit{check}(k, s, f1) \vee \mathit{check}(k, s, f2))$ %(8, 9)%
- $\mathit{check}(k, s, EU(f1, f2)) \Leftrightarrow (\exists c : \mathit{Path} \cdot (c\ \mathit{eps}\ \mathit{paths}(s, k) \wedge$
  $\exists j : \mathit{Nat} \cdot (j < \#(\mathit{allPathStates}(c, k)) \wedge (\mathit{check}(k, \mathit{allPathStates}(c, k)!j, f2) \wedge$
  $\forall l : \mathit{Nat} \cdot (l < j \Rightarrow \mathit{check}(k, \mathit{allPathStates}(c, k)!l, f1))))))$ %(10, 11)%
- $\mathit{check}(k, s, AU(f1, f2)) \Leftrightarrow$
  $\mathit{check}(k, s, \mathit{ANDCTL}(\mathit{NOTCTL}(EU(\mathit{NOTCTL}(f2),$
  $\mathit{ANDCTL}(\mathit{NOTCTL}(f1), \mathit{NOTCTL}(f2)))),$
  $\mathit{NOTCTL}(EG(\mathit{NOTCTL}(f2)))))$ %(12, 13)%
- $\mathit{check}(k, s, EF(f1)) \Leftrightarrow \mathit{check}(k, s, EU(TT, f1))$ %(14, 15)%
- $\mathit{check}(k, s, AF(f1)) \Leftrightarrow$
  $\mathit{check}(k, s, \mathit{NOTCTL}(EG(\mathit{NOTCTL}(f1))))$ %(16, 17)%
- $\mathit{check}(k, s, EX(f1)) \Leftrightarrow$
  $(\exists t : \mathit{Nat} \cdot (\mathit{isNext}(t, s, k) \wedge \mathit{check}(k, t, f1)))$ %(18, 19)%
- $\mathit{check}(k, s, AX(f1)) \Leftrightarrow$
  $\mathit{check}(k, s, \mathit{NOTCTL}(EX(\mathit{NOTCTL}(f1))))$ %(20, 21)%
- $\mathit{check}(k, s, EG(f1)) \Leftrightarrow (\exists c : \mathit{Path} \cdot (c\ \mathit{eps}\ \mathit{paths}(s, k) \wedge$
  $(\exists n : \mathit{Nat} \cdot (n < \#(\mathit{allPathStates}(c, k)) \wedge$
  $(\exists l : \mathit{Nat} \cdot (l \leq n \wedge \mathit{isNext}(\mathit{allPathStates}(c, k)!l, \mathit{allPathStates}(c, k)!n, k) \wedge$
  $(\forall r : \mathit{Nat} \cdot (r \leq n \Rightarrow \mathit{check}(k, \mathit{allPathStates}(c, k)!r, f1)))))))))$ %(22, 23)%
- $\mathit{check}(k, s, AG(f1)) \Leftrightarrow$
  $\mathit{check}(k, s, \mathit{NOTCTL}(EF(\mathit{NOTCTL}(f1))))$ %(24, 25)%

Fig. 5. The CTLOperators Specification (Axioms).

activities and their relationships in terms of artifacts flow are illustrated in Figure 6. The validation process consists of continuous feedback of these activities through specification analysis and test results analysis of conformance of the IUT to the specification. The lack of conformance may reveal either an IUT defect or specification inconsistency (not all tests can pass at the same time with conflicting test results). Moreover, uncovered requirements and surprises (unpredictable behaviour) may be detected. Therefore, the process is iterative and incremental and it is performed until planned coverage is achieved and all tests pass.

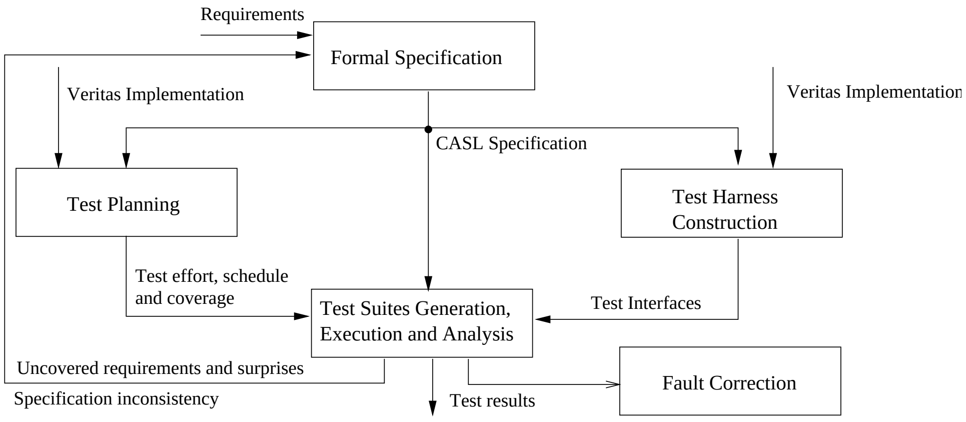
Requirements
Formal Specification
Veritas Implementation
Veritas Implementation
CASL Specification
Test Planning
Test Harness Construction
Test effort, schedule and coverage
Test Suites Generation, Execution and Analysis
Test Interfaces
Uncovered requirements and surprises
Specification inconsistency
Test results
Fault Correction

Fig. 6. Validation process followed in the case study.

Test planning.

The objective is to determine: who performs the tests, which pieces will be tested, when will tests be performed and how much testing is adequate (coverage metrics to be achieved) [23]. In the VERITAS case study, no member of the testing team has participated in the application implementation to maximise the chances of finding defects in the implementation. This team has also being responsible for constructing the algebraic specification that is the basis for test suite generation. Concerning which pieces to be tested, we opt for testing CTL formulas evaluation and kripke structure properties. Tests are organised in blocks according to the structure of the specification and performed as soon as the block is complete and necessary test harness is available. According to Definition 2.3, the tests to be performed are as follows.

$$\bigwedge_{(\psi,T)\,\in\,\Psi_{CTLOperators}} A, \sim_A, \doteq_A \models^T \psi$$

$$\bigwedge_{(\psi,T)\,\in\,\Psi_{Path}} A|_{Path}, \sim_{A|_{Path}}, \doteq_{A|_{Path}} \models^T \psi$$

$$\bigwedge_{(\psi,T)\,\in\,\Psi_{Kripke}} A|_{Kripke}, \sim_{A|_{Kripke}}, \doteq_{A|_{Kripke}} \models^T \psi$$

where $\Psi_{CTLOperators}$, Ψ_{Path}, Ψ_{Kripke}, represent the axioms introduced in the CTLOPERATORS, PATH and KRIPKE specifications respectively, A is an implementation matching the signature of CTLOPERATORS, $A|_{Path}$, $A|_{Kripke}$ is the implementation A restricted to the signatures of PATH and KRIPKE respectively, and $\sim_{A|_{Path}}$, $\sim_{A|_{Kripke}}$ are approximate equalities restricted to the signatures of PATH and KRIPKE respectively. Note that we assume the implementation of lists, sets and strings in SML are sound. Therefore, we are not considering testing from the List, Set and String specifications.

Finally, we considered coverage of specification axioms and equivalence partition on test data along with modified decision condition coverage of positive decisions in axioms [17] as the main metrics to be achieved. Nevertheless, unreachable data should be tried to assess robustness.

Formal Specification.

SBT is strongly based on the existence of a formal specification that is reasonably complete, consistent and correct. This justifies the need for an iterative and incremental validation process, where the specification can be improved. In our case study, we constructed an algebraic specification of a generic model checker with the desired behaviour for the application, in consultancy with model checking experts and developers. The specification has been refined for minor defects w.r.t. to requirements and missing requirements (details in Section 5 – Running Tests and Analysing Results). Part of the specification produced is presented in Section 3. This specification has been parsed and type-checked using the CATS system v0.88 [24].

Test Harness Construction.

Test harness is the necessary support to run the tests such as test drivers, data generators and IUT test interfaces. In our case study, test oracles are the test drivers generated in a semi-automatic way along with data using the CaslTest tool [26,25].

Test interfaces may be required to allow test drivers access to specific functions of the IUT that are not directly exported. This is due to:

(i) The structure of the specification may differ from the structure of the IUT. For instance, Casl specifications may have different names and signature when compared to the IUT;

(ii) The specification may have auxiliary operations that are not implemented in the IUT. In this case, additional code may be needed for testing purposes only;

(iii) Standard types in the specification language are usually mapped to corresponding ones in the programming language and we assume their implementation is sound. However, Casl library functions may have different names and slightly different signatures when comparing to the corresponding ones in SML;

(iv) The specification may not provide constructors for a given sort.

For 1 and 3, we constructed a *facade*[13] structure that translates names and adapts function calls, based on the translation specification-building operation [21]. The *facade* is extended with constructors for handling 4. Test oracles calls are redirected from the *facade* structure to the functionality of interest in the IUT. As this is a controlled specification transformation, it should bring no suspicion to the process. However, 2 is more critical: the additional code requires a rigorous verification process to avoid false fault detection and fault disguise in the IUT.

Test Suites Generation, Execution and Analysis.

We followed the approach proposed in [20,22,26,25] that consists of activities commonly found on a generic testing process where the grey-box approach is used to test oracle generation. The approach has also guidelines to test case and test data selection as well as running tests and analysing results.

Testing from algebraic specifications boils down to checking whether specification axioms are satisfied by an IUT – a module that exports a number of operations according to a given signature [15]. 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. A *test* is the process of executing an operation of the IUT instantiated with a particular set of values.

The generic testing model to be used consists of the following activities: test case selection[14], test data selection[15], test oracle generation, test execution and interpretation of results. *Test cases* are extracted from formal specifications together with *test data sets* which are defined at specification level and associated with each test case. Then, oracles are defined for each test case. An oracle is a predicate to evaluate a test case according to test results, incorporating procedures to compute equality on non-observable sorts. For each test case, the corresponding oracle is run at testing execution. Basically, this consists in computing each test involved in the oracle predicate and combining results according to the predicate for each possible combination of values from the selected test data. Note that test data is mapped to actual corresponding values in the IUT. Once all tests are computed, oracles return a verdict that is interpreted according to the grey-box approach in terms of presence/absence of errors.

[13] Design pattern that defines a unified higher-level interface that makes a system easier to use.

[14] A test case is a statement about what a test covers (input criteria, acceptance criteria), that can be expressed by logical predicates or even by informal statements.

[15] A test data or test set is an instance of a test case which consists of a collection of inputs submitted to a program in order to test it according to the test case.

Test Hypotheses.

SBT is based on hypotheses concerning the IUT and the specification so that conclusions can be reached from test results [15]. In our approach, the main hypotheses are: (1) Observable sorts implementation in SML is sound; (2) Test harness implementation is sound; (3) Test data selection is based on uniformity hypothesis, i.e., only one value per subdomain is required. Subdomain definition is guided by the specification [15].

Our goal is to avoid rejecting correct programs in the sense of Theorem 2.2 (Section 2). In case axioms do not meet quantifiers constraint, unbiased test sets are required. It is important to note that we assume SML programs are modelled as algebras.

5 The Case Study Results

This section describes the case study experience according to the activities of the validation process presented in Section 4, focusing on the general procedures followed, the main results achieved, obstacles faced and solutions used. These are illustrated by examples.

Formal Specification.

The Casl specification presented in Section 3 has been developed with the purpose of validating the Veritas model checker. The benefits of constructing the specification, even though after Veritas first release, have been considerable, other than providing the basis for a SBT process. A better understanding of Veritas behaviour among the research team has been reached. The specification process has promoted a broad discussion on checking algorithms that lead to improvements on the tool documentation as a whole. For instance, how to deal with the notion of infinite paths in a kripke structure, when checking the `EG` operator (Axiom (22,23) in Figure 5).

Test Case Selection.

When testing from algebraic specifications, we aim at checking whether axioms are satisfied by programs. Axioms express properties in terms of operation applications and equality of values produced, combined by logical connectives. Therefore, test cases are directly generated from axioms, since they indeed characterise a scenario or combination of scenarios of execution of operations we want to check. For simplicity, each axiom is regarded here as a separate test case. However, due to the grey-box approach, whenever quantified formulas occur as one side of *iff* predicates, the corresponding axiom needs to be split into two: one for each direction of the implication. The reason is that, in the presence of *iff*, equations and quantifiers can be classified as both positive and negative since there are two directions of implication to be considered.

Test cases selected, considering the specifications presented in this paper, are labelled in Figures 2, 3 and 5 as `(i)`, where `i` is an ordering number. Label `(i,j)` means that two test cases are indicated with `i` being the implication from left to right and `j` from right to left. A total of 48 test cases have been selected.

Test Data Selection.

SBT data 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 and test data can be derived as specifications are created. Test sets are defined here as sets of ground terms built by successively combining specification symbols (constants and operations) [20] which are subsequently translated to sets of values in the IUT. A good test set is usually defined as one that it is cost-effective and reveals faults [1]. For this, an effective selection technique must be applied. Also, given a test case, a set of data is selected so that an adequate coverage is achieved.

The abstraction gap between specification and IUT can create barriers to generate test data based on specifications only. For instance:

(i) Specifications may not have (explicit) constructors for its sorts (see Kripke specification and the *Kripke* sort);

(ii) IUT may have *junks*, unreachable values in the target data type, not generated by any (constructor) term (see the *Nat* sort that can be implemented as the type *int* in SML, for simplicity);

(iii) IUT may have confusion, one or more terms that are translated to the same concrete value. This may lead to redundant data (see the CTLOperators specification where equivalent formulas can be built by different operators combination);

(iv) Real values may be too complex to be denoted by ground terms (see Kripke specification and the *Kripke* sort);

(v) Data implementation may have different subdomains from the specification ones.

To handle 1 and 5, we opt to consider concrete data from the IUT for the *kripke* sort. Also, we have to be concerned with maximum memory capacity to deal with a single value of this sort: a real kripke structure has a great number of states and connections between them. For the test cases we selected, we concluded that in average we could afford only one kripke structure at a time, considering only real structures in spite of toy ones. To work with more than one kripke structure, they have to be considered in different test executions, using the same test harness. The *Kripke* sort is implemented as the `ss_table` type presented in Figure 7, where the *State* sort is implemented as the `ss_node` type. In the tests, we considered kripke structures with 17 to 1094 nodes.

```
type ss_node = {
    id : int, node_struct : string list,
    node_msgs : string list, node_events : string list,
    node_suc : int list, node_pred : int list };
type ss_table = (int, ss_node) Polyhash.hash_table;
```

Fig. 7. `ss_table` type that implements the *Kripke* sort.

To handle point 2 mentioned above, we focused on specification generated values only, but also considered limit unreachable values (e.g. negative integers) to assess IUT robustness. To handle 3, we opted to work with simple formulas, covering possible basic combinations of constructors with at most 2 constructors per formula. Finally, to handle 5, we analysed subdomains selected at specification level and compared to code level subdomains and added relevant data to the test sets.

For data rather than kripke structures, we use the following selection criteria: (1) exhaustiveness with respect to the kripke structure being considered, when feasible; (2) otherwise, random choice of values that guarantee MC/DC coverage of truth decisions [10] and unbiasedness; (3) choice of values in the limit of partitions of data not related to the kripke structure being considered. As an example, consider Test Case 22 (Figure 5) where the existential quantifier appears in positive positions. In this case, three unbiased test sets of natural numbers and one unbiased test set of paths are required, i.e., these test data must include the witnesses for `c`, `n` and `l`, if they exists in the absolute exhaustive test sets, that makes the right hand side of the axiom be true when the left hand side is true, for the kripke structure and the formulas being considered. Trivial unbiased test sets are the sets of all paths and all indexes of these paths. However, when they cannot be used (too big or unfeasible), it is important to eliminate redundant data and be more selective. Unbiasedness must be checked to avoid rejecting correct IUTs.

Finally, it is important to remark that, according to test planning and based on Theorem 2.4, we define test sets for each axiom in the scope of the signature where the axiom is declared.

Test Oracle Generation.

Test oracles have been generated for each test case selected according to the grey-box approach, using the CaslTest tool [26,25]. The main concern is to identify positive and negative contexts and apply sound/complete equalities for non-observable sorts according to Theorem 2.2. We considered *Kripke*, *State*, *Path*, *CTLF* and *PROPEVAL* as non-observable sorts.

Figure 8 shows the oracle for Test Case 12 from the Kripke specification, where `LNat0` and `LNat1` are the identifiers of lists of values of *Nat* representing state ids (test set), `LKripke2` is a list of *Kripke* values and `forall` is an implementation of the ∀ quantifier. Notice that *State* is a non-observable sort. Thus equality on values of this sort, occurring in a negative position, is computed by a sound equality named `eqWSTATE` that it is implemented in the `FacadeKripke` structure according to the internal representation of *State* (See `ss_node` in Figure 7). From Theorem 2.4, equalities must be defined in the scope of the signature being considered.

It is important to remark that, since test oracle code is automatically generated, we can assume their implementation is sound, resulting in significant gains in productivity and confidence in the results of running the tests. This is crucial, if we want to cope with changes: test code must evolve as IUT and specifications do.

```
fun oracle( ) =
  forall LNat0(fn s1 ⇒
    forall LKripke2(fn k ⇒
      forall LNat1(fn s2 ⇒
        if s1 eps allStates(k) andalso s2 eps allStates(k) then
          if eqWSTATE(getState(s1,k), getState(s2,k)) then
            s1= s2
          else true
        else true)))
```

Fig. 8. Test oracle for Test Case 12 from the Kripke specification.

Test Harness.

According to test planning, we constructed 3 SML structures to act as test interface for the test oracles to be run, one for each of the specifications presented in Section 3: `FacadeKripke`, `FacadePath` and `FacadeCTL`. Generally, these structures contain direct function calls demanded by the need to translate names from Casl libraries to SML library names. For instance, Set in Casl is translated to *Binaryset* in SML. The translation has been done according to rules presented in [21] and reviews have been performed to check them. The test interfaces also include functions that generate the test data sets in SML, according to the specification of these data. For instance, the list of state identifiers for a given kripke structure (see *Kripke* specification). Again, these are mostly direct function calls to constructors and predefined functions in SML.

The real challenge is to provide an implementation for specification hidden symbols, i.e., specification operations not implemented in the IUT. Special care is needed to avoid including faults in the test interfaces that could lead to the rejection of a correct IUT. For instance, this problem arises in the *Path* specification – the *paths* operation. Obviously, it is impossible to implement such a function as it is specified. However, *EG* and *EU* operators depend on finding a path among all paths. Rather than constructing the set of all paths, VERITAS constructs paths *on-the-fly* when checking formulas that involves such operators. Nevertheless, this function is needed for testing purposes.

The *paths* function has been implemented in the `FacadePath` structure according to the path constraints in Axiom (22,23) of the CTLOPERATORS specification – a set of finite paths that is computed by detecting loops. This function has been checked according to test cases generated from the PATH specification using the test process presented in this paper (see Definition 2.3 and the hide operation). This has been done as tests for the PATH specification were conducted.

A further obstacle is to check the path construction procedure used by VERITAS according to axioms of the PATH specification. The problem is that it is impossible to compute a path from function calls. There is not such a function as paths are constructed as needed when *EG* and *EU* operators are considered. In this case, we decided to have path test data sets composed of paths generated as example and counterexample by VERITAS. These data has been collected by running model checking experiments using the tool and the kripke structured being considered. In this way, we managed to check indirectly the paths constructed by the tool.

Furthermore, the process of building the test interfaces has been beneficial to improve the VERITAS documentation. It has promoted a thorough investigation of interfaces and internal algorithms to guarantee the right functions are called with the right parameters, specially, functionality related to path construction. The final test code produced has 3.233 lines, where 2.895 was automatically generated by the CASLTEST tool (test oracles). The version of VERITAS considered has 840 lines of code, excluding predefined Moscow ML libraries used[16], giving the expected rate of the test code at about 3 times bigger than the IUT [23]. The test code can be reused to run regression tests

[16] http://www.dina.dk/~sestoft/mosml.html

in the application.

Running Tests and Analysing Results.

As described in Section 4, test oracles for each specification are run in different experiments. There are three possible answers: 1) succeeded; 2) failed; 3) raised an exception. For 1), we can not conclude the IUT satisfies the axiom, but we can increase our confidence by analysing the quality of the test data selected and intended coverage. According to the grey-box approach and the purpose to detect incorrectness followed, 2) is conclusive – there is a fault in the IUT. Even with this guarantee, we also analysed specification and manually produced harness before reaching the conclusion. Finally, 3) may indicate unpredicted behaviour or partial handling of data, for instance, unreachable values.

The validation process occurred in an iterative and incremental way. We begun with an initial specification and gradually test harness was constructed as a deeper understanding of the application was gained. When running the tests, minor defects were found in the specification as a result of analysing test failures and exceptions raised, such as $\leq$ in place of $<$ and uncovered preconditions as $\mathit{s1}\ \mathit{eps}\ \mathit{allStates}(k)$ (see Figure 2). Part of the final specification is presented in Section 3. We also uncovered missing requirements and surprising behaviour in the IUT (developers were not aware of) from test results. They were analysed by the team that decided to incorporate them or not in the specification as new axioms. For instance, the treatment of invalid State identifiers (See *Kripke* specification). This has been achieved by thorough test data selection.

Furthermore, tests that could not be implemented often reviewed functionality misconception or a huge gap between specification and the IUT. For instance, we could opt to focus on either abstract CTL semantics or model checking of properties expressed in CTL, having atomic propositions as labels of a state. The more abstract is the specification the hardest is harness construction. The less abstract the biggest the risk of overspecifying and loose sight of the test perspective.

At the end, the version of the VERITAS under test has passed all test cases with the intended coverage achieved. Although not an absolute guarantee of conformance to the specification, this greatly increases confidence that the implementation satisfies its formalised requirements. Test hypotheses have been carefully considered throughout the processed even though not proved. Prior to the final iteration, the test cases generated revealed faults that lead to improvements in the code.

Strengthening confidence on test hypotheses.

All observable sorts were mapped to predefined types or exported libraries. Test harness has been mostly automatically generated and the manually produced part has been rigorously verified. Moreover, subdomains of data in the code have been analysed and taken into account for data selection.

The core of Veritas implementation is focused on the functional part of the SML language, avoiding collateral effects. Input and output operations are used for reporting. This is also a single-threaded applications, making it suitable for algebraic SBT from Casl specifications.

Lessons Learned.

For SBT, the main lessons learned are:

- The grey-box approach is practical to analyse test results in cases of success and failure. However, as expected, it gives no clue to pinpoint a fault, other than detecting its existence. Statical analysis must not be discarded in the validation process considered;
- Producing a specification at the right abstraction level is crucial to SBT;
- Dealing with hidden symbols is far more complicated than simply have to implement additional code. They tend to be difficult to implement and verify, but cannot be avoided without the danger of overspecifying;
- Apart from oracle generation, full test harness cannot be fully automated. It is likely that coding is inevitable. In this case, it is crucial to follow patterns of implementation;
- SBT is strongly based on hypotheses, but they can be reasonably confirmed;
- The process of producing a complete, consistent and correct specification is hard. In this sense, SBT can be beneficial to the long term process of constructing and maintaining formal specifications, since it represents a bridge between specification and IUT that can provide feedback on feasibility and focus on the real requirements to be met.

6 Concluding Remarks

A case study on algebraic specification based-testing is presented. Issues related to testing SML implementations against structured algebraic specifications are discussed, considering a generic model of the testing process and theoretical solutions proposed in the literature. The case study – the Veritas model checker – has been checked according to a proposed methodology. In this methodology, a specification in Casl is constructed. From this specification, test cases are selected and test oracles automatically generated. Special care is taken on data selection and test harness construction. This paper also discusses advantages and limitations of the approach followed, including benefits of specification-based testing on validating code and specification by providing a bridge between them.

The tests produced can be applied to check further versions of Veritas and even other similar model checkers w.r.t. to the Casl specification. This is particularly important for this application since a process of algorithms optimisation is under development. Also, they can be applied to the new tools that have been developed by the local research team to cooperate with Veritas, for instance, a space state generator. Finally, test oracles can be easily generated again to cope with changes in the specification.

The time required for the case study was about 2 months with a team of 3 researchers. During this time, faults have been detected and artifacts improved as a result of the solutions applied. As further work, we aim to use the experience of this case study to improve current processes and tools for SBT such as CaslTest and deal with generacity and modularity of specifications as well as subtyping and exception handling. Patterns of test harness design and implementation also deserves further investigation.

References

[1] Beizer, B., "Black-Box Testing: Techniques For Functional Testing of Software and Systems," John Wiley & Sons Inc., 1995.

[2] Belinfante, A., J. Feenstra, L. Heerink and R. G. de Vries, *Specification based formal testing: The easylink case study*, in: *Progress 2001 2nd workshop on Embedded Systems*, Veldhoven, the Netherlands, 2001, pp. 73 – 82.

[3] Bernot, G., M.-C. Gaudel and B. Marre, *Software testing based on formal specifications: a theory and a tool*, Software Engineering Journal **6** (1991), pp. 387–405.

[4] Bicarregui, J., J. Dick, B. Matthews and E. Woods, *Making the most of formal specification through animation, testing and proof*, Science of Computer Programming **29** (1997), pp. 53–78.

[5] Bidoit, M., M. V. Cengarle and R. Hennicker, *Proof systems for structured specifications and their refinements*, in: *Algebraic Foundations of Systems Specifications*, IFIP State-of-The-Art Reports, Springer, 1999 pp. 385–433.

[6] Bidoit, M. and P. D. Mosses, "Casl User Manual," Lecture Notes in Computer Science (IFIP Series) **2900**, Springer, 2004, with Chapters by T. Mossakowski, D. Sannella and A. Tarlecki.

[7] Carrington, D. and P. Stocks, *A tale of two paradigms: Formal methods and software testing*, in: J. E. Nicholls and J. A. Halls, editors, *Z User's Meeting* (1994), pp. 51–68.

[8] Cheesman, J. and J. Daniels, "UML Components – A Simple Process for Specifying Component-Based Software," Component Software Series, Addison-Wesley, 2001.

[9] Chen, H. Y., T. H. Tse, F. T. Chan and T. Y. Chen, *In black and white: an integrated approach to class-level testing of object-oriented programs*, ACM Transactions on Software Engineering Methodology **7** (1998), pp. 250–295.

[10] Chilenski, J. J. and S. P. Miller, *Applicability of modified condition/decision coverage to software testing*, Software Engineering Journal **7** (1994), pp. 193–200.

[11] Clarke, E. M., O. Grumberg and D. E. Long, *Model checking and abstraction*, in: *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1992), pp. 343–354.

[12] Clarke, E. M., O. Grumberg and D. A. Peled, "Model Checking," MIT Press, 1999.

[13] Doong, R. and P. G. Frankl, *The ASTOOT approach to testing object-oriented programs*, ACM Trans. on Software Engineering and Methodology **3** (1994), pp. 101–130.

[14] Dwyer, M. B., G. S. Avrunin and J. C. Corbett, *Patterns in property specifications for finite-state verification*, Technical Report UM-CS-1998-035, University of Massachusetts (1998).

[15] Gaudel, M.-C., *Testing can be formal, too*, in: P. D. Mosses, M. Nielsen and M. I. Schwartzbach, editors, *Proceedings of Theory and Practice of Software Development - TAPSOFT'95*, Lecture Notes in Computer Science **915** (1995), pp. 82–96.

[16] Guerrero, D. D. S., *Redes de petri orientadas a objeto*, Tese de Doutorado - COPELE, UFCG (2002).

[17] Hayhurst, K. J., D. S. Veerhusen, J. J. Chilenski and L. K. Rierson, *A practical tutorial on modified condition/decision coverage*, Technical Report TM-2001-210876, NASA (2001).

[18] Hennicker, R., "Structured Specifications with Behavioural Operators: Semantics, Proof Methods and Applications," Habilitation thesis, Institut fur Informatik, Ludwig-Maximillians-Universitat Munchen, Munchen, Germany (1997).

[19] Hoare, C. A. R., *How did software get so reliable without proof ?*, in: M.-C. Gaudel and J. Woodcock, editors, *Formal Methods Europe - FME'96*, Lecture Notes in Computer Science **1051** (1996), pp. 1–17.

[20] Machado, P. D. L., *On oracles for interpreting test results against algebraic specifications*, in: A. M. Haeberer, editor, *Algebraic Methodology and Software Technology, AMAST'98*, Lecture Notes in Computer Science **1548**, Springer, 1999 pp. 502–518.

[21] Machado, P. D. L., *Testing from structured algebraic specifications*, in: T. Rus, editor, *Alg. Method. and Soft. Technology, 8th Int. Conference, AMAST'00*, Lecture Notes in Computer Science **1816** (2000), pp. 529–544.

[22] Machado, P. D. L., "Testing from Structured Algebraic Specifications: The Oracle Problem," Ph.D. thesis, LFCS - Division of Informatics, University of Edinburgh, UK (2000), also LFCS Technical Report 00-423, http://www.lfcs.informatics.ed.ac.uk/reports/.

[23] McGregor, J. D. and D. A. Sykes, "A Practical Guide to Testing Object-Oriented Software," Object Technology Series, Addison-Wesley, 2001.

[24] Mossakowski, T., *CASL: From semantics to tools*, in: S. Graf, editor, *TACAS'2000*, Lecture Notes in Computer Science **1785** (2000), pp. 93–108, also, http://www.informatik.uni-bremen.de/cofi/Tools/index.html.

[25] Oliveira, K. A., P. D. L. Machado and W. L. Andrade, *Casltest – test case, test oracle and test data generation from casl specifications*, in: *XVII Brazilian Symposium on Software Engineering - SBES'2003 - Tools Session*, Manaus, Brazil, 2003, pp. 73–78.

[26] Pinto, A. L. S., K. A. Oliveira and P. D. L. Machado, *Automating formal testing from casl specifications*, in: *IV Workshop on Formal Methods*, Rio de Janeiro, 2001, pp. 37–48.

[27] Rodrigues, C. L., "Verificação de Modelos em Redes de Petri Orientadas a Objetos," Master's thesis, Pós-Graduação em Informática - UFCG (2004).

[28] Rodrigues, C. L., P. E. S. Barbosa, D. D. S. Guerrero and J. C. A. de Figueiredo, *Rpoo model checker*, in: *Tools Session - SBES'2004*, Brasília - Brasil, 2004, pp. 79–85.

[29] Sannella, D., S. Sokołowski and A. Tarlecki, *Toward formal development of programs from algebraic specifications: parameterisation revisited*, Acta Informatica **29** (1992), pp. 689–736.

[30] Valmari, A., *The state explosion problem.*, Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models **1491** (1998), pp. 429–528.

[31] Wirsing, M., *Algebraic specification*, in: J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Elsevier Science Publishers, 1990 pp. 675–788.