Unit Testing for CASL Architectural Specifications

Patricia D.L. Machado[1] and Donald Sannella[2]

[1] Systems and Computing Department, Federal University of Paraiba
[2] Laboratory for Foundations of Computer Science, University of Edinburgh

Abstract. The problem of testing modular systems against algebraic specifications is discussed. We focus on systems where the decomposition into parts is specified by a CASL-style architectural specification and the parts (*units*) are developed separately, perhaps by an independent supplier. We consider how to test such units without reference to their context of use. This problem is most acute for generic units where the particular instantiation cannot be predicted.

1 Introduction

Formal testing is concerned with deriving test cases from formal specifications and checking whether programs satisfy these test cases for a selected finite set of data, as a practical alternative to formal proof. Much work has focused on test case selection and (automatic) interpretation of test results. Testing from algebraic specifications has been investigated, focusing on both "flat" specifications [Ber89,Gau95,LA96,Mac99] and structured specifications [LA96,Mac00b,DW00]. It is often assumed that the structure of the specification matches the structure of the program [LeG99,DW00] although it is possible to take specification structure into account without making this strong assumption [Mac00b].

Tests are not interesting unless we can interpret their results in terms of correctness. *Oracles* are decision procedures for interpreting the results of tests. The *oracle problem* arises whenever a finite executable oracle cannot be defined; this may stem e.g. from the semantic gap between specification and program values. Guaranteeing correctness by testing requires tests covering all possible interactions with the system, usually an infinite and impractical activity. When only finite sets of interactions are considered, successful testing can accept incorrect programs and unsuccessful testing can reject correct programs. The latter must be avoided since the costs of finding and fixing errors are too high to waste effort on non-errors. Accepting incorrect programs is not a problem as long as test coverage is adequate, since testing is not expected to guarantee correctness.

This paper addresses testing of modular systems where parts are developed independently from CASL-style architectural specifications [ABK+03,BST02]. In this context, the problem reduces to that of testing units independently in such a way that their integration can be checked in a cost-effective way. An architectural specification consists of a list of *unit declarations*, naming the units (components)

that are required with specifications for each of them, together with a *unit term* that describes the way in which these units are to be combined.

When testing modular systems and their parts, different perspectives need to be considered. From a *supplier's* point of view, units have to be checked independently of their contexts of use. Checking generic units poses special problems since the particular instantiation is unknown in general, and the set of all possible instantiations is almost always infinite. In this paper, we present styles of testing non-generic and generic units that address these problems. The main novelty is in the case of generic units. *Integration testing* is left to future work. A extended version of this paper with examples demonstrating most of the main definitions and results is available as an Edinburgh technical report.

2 Preliminary definitions

Algebraic Specifications. We assume familiarity with basic concepts and notation of algebraic specification, where programs are modelled as algebras. Each signature $\Sigma = (S, F, Obs)$ is equipped with a distinguished set $Obs \subseteq S$ of *observable sorts*. $Sign$ is the category of signatures. We consider specifications with axioms of first-order equational logic, and write $Sen(\Sigma)$ for the set of all Σ-sentences. We restrict to algebras with non-empty carriers, and write $Alg(\Sigma)$ for the class of all such Σ-algebras. $[A]$ denotes the reachable subalgebra of $A \in Alg(\Sigma)$.

Behavioural Equality. For a non-observable sort, it is not appropriate to assume that equality on values is the usual set-theoretical one. Equality on values of a Σ-algebra A can be interpreted by an appropriate *behavioural equality*, a partial Σ-congruence $\approx_A = (\approx_{A,s})_{s \in S}$ with $Dom(\approx_A) = \{a \mid a \approx_A a\}$. 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.[3] A *Σ-behavioural equality* is defined as $\approx_\Sigma = (\approx_{\Sigma,A})_{A \in Alg(\Sigma)}$, one behavioural equality for each Σ-algebra A.

Approximate Equality. Behavioural equality can be difficult to test. Consider e.g. observational equality on non-observable sorts, defined in terms of a set of contexts that is usually infinite. One approach involves the use of *approximate equalities* [Mac99,Mac00c] which are binary relations on values of the algebra. An approximate equality is *sound* with respect to a behavioural equality if all values (in $Dom(\approx_A)$) that it identifies are indeed equal, or *complete* if all equal values are identified. Any *contextual equality* $\sim_{\mathcal{C},A}$ defined from a subset $\mathcal{C}$ of the observable computations[4] is complete with respect to observational equality, although it need not be a partial congruence. The set-theoretical equality is sound – in programming terms, this is equality on the underlying data representation.

[3] 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 sort s are *observationally equal*, $a \approx_{Obs,A,s} b$, iff $a, b \in [A]$ and $\forall C \in C_{Obs} \cdot \forall \alpha : X \to [A] \cdot \alpha_a^\#(C) = \alpha_b^\#(C)$, where $\alpha_a, \alpha_b : X \cup \{z_s\} \to [A]$ are the extensions of α defined by $\alpha_a(z_s) = a$ and $\alpha_b(z_s) = b$.

[4] Let $\mathcal{C} \subseteq C_{Obs}$. Values a and b are *contextually equal w.r.t* $\mathcal{C}$ iff $a, b \in [A]$ and $\forall C \in \mathcal{C} \cdot \forall \alpha : X \to [A] \cdot \alpha_a^\#(C) = \alpha_b^\#(C)$. Obviously, if $\mathcal{C} = C_{Obs}$, then $\sim_{\mathcal{C},A} = \approx_{Obs,A}$.

Families of Equalities. The signature of different parts of a structured specification may differ, and the interpretation of equality (e.g. observational equality) may depend on the signature. So, to deal with structured specifications, we use *families* of equalities indexed by signatures. Let $\approx = (\approx_\Sigma)_{\Sigma \in Sign}$ be a family of behavioural equalities and let $\sim = (\sim_\Sigma)_{\Sigma \in Sign}$ and $\doteq = (\doteq_\Sigma)_{\Sigma \in Sign}$ denote arbitrary families of approximate equalities. The family $\sim$ is complete (sound) w.r.t. $\approx$ if $\sim_\Sigma$ is complete (sound) w.r.t. $\approx_\Sigma$ for all Σ. The *reduct* of $\sim_\Sigma$ by $\sigma : \Sigma' \to \Sigma$ is $(\sim_\Sigma)|_\sigma = ((\sim_{\Sigma,A})|_\sigma)_{A \in Alg(\Sigma)}$ where $(\sim_{\Sigma,A})|_\sigma = ((\sim_{\Sigma,A})_{\sigma(s)})_{s \in S'}$; then the family $\sim$ is *compatible* with signature morphisms if for all $\sigma : \Sigma' \to \Sigma$ and all $A \in Alg(\Sigma)$, $\sim_{\Sigma', A|_\sigma} = (\sim_{\Sigma,A})|_\sigma$. For the results below having compatibility as a condition, it is enough to consider the signatures arising in the structure of a given specification (for a normalized specification, this includes intermediate signatures arising during normalization). Compatibility may fail if these signatures have different sets of observers for the same sort. The family of set-theoretical equalities is always compatible, but it is easy to check that the family of observational equalities is not compatible in general. One may restrict the introduction of new observers to avoid this problem, see e.g. [BH99].

3 Formal Testing from Algebraic Specifications

Testing from algebraic specifications boils down to checking if axioms are satisfied by programs [Gau95]. Oracles are usually active procedures which drive the necessary tests and interpret the results. *Test cases* are extracted from specifications together with *test sets* defined at specification level and associated with axioms. Then, oracles are defined for each test case or group of test cases. A *test obligation* combines a test case, test data, a test oracle and a program to be tested. When there is no chance of confusion, this is referred to simply as a *test*.

Test Cases and Test Data. For simplicity, each axiom is regarded as a separate test case. Techniques exist to simplify test cases or make them more practical, but this is a separate topic. Test data sets are usually defined from specifications rather than from programs. Here, test sets will be taken to be sets of ground terms [Gau95,Mac99] corresponding to sets of values in the program under test.

Test Oracle Design. The oracle problem often reduces to the problem of comparing values of a non-observable sort for equality; when equality is interpreted as behavioural equality, e.g. observational equality, it may be difficult or impossible to decide. Quantifiers ranging over infinite domains make the oracle problem more difficult. An approach to defining oracles that addresses these problems is presented in [Mac99], where equality is computed using two approximate equalities, one sound and one complete. These equalities are applied according to the context in which equations occur, positive or negative[5]. To handle the quantifier

[5] 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.

problem, restrictions are placed on the contexts in which they can occur. An *approximate oracle* is then a procedure that decides whether a given axiom is satisfied by a program or not. Such an oracle computes a "testing satisfaction" relation (given below) which differs from the standard one in the way equality is computed and also because quantifiers range only over given test sets.

Definition 3.1 (Testing Satisfaction). *Let Σ be a signature, $T \subseteq T_\Sigma$ be a Σ-test set, $\sim, \mathrel{\dot{=}}$ be two Σ-approximate equalities, A be a Σ-algebra and $\alpha : X \to Dom(\approx_A)$ be a valuation.* Testing satisfaction $\models^T_{\sim,\mathrel{\dot{=}}}$ *is defined as follows.*

1. $A, \alpha \models^T_{\sim,\mathrel{\dot{=}}} t = t'$ *iff* $\alpha^\#(t) \sim_A \alpha^\#(t')$;
2. $A, \alpha \models^T_{\sim,\mathrel{\dot{=}}} \neg\psi$ *iff* $A, \alpha \models^T_{\mathrel{\dot{=}},\sim} \psi$ *does not hold;*
3. $A, \alpha \models^T_{\sim,\mathrel{\dot{=}}} \psi_1 \wedge \psi_2$ *iff both* $A, \alpha \models^T_{\sim,\mathrel{\dot{=}}} \psi_1$ *and* $A, \alpha \models^T_{\sim,\mathrel{\dot{=}}} \psi_2$ *hold;*
4. $A, \alpha \models^T_{\sim,\mathrel{\dot{=}}} \forall x{:}s \cdot \psi$ *iff* $A, \alpha[x \mapsto {}^\#(t)] \models^T_{\sim,\mathrel{\dot{=}}} \psi$ *holds for all* $t \in T_s$

where $\alpha[x \mapsto v]$ is the valuation α superseded at x by v. Satisfaction of formulae involving $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\exists$ is given using their definitions in terms of $\neg$, $\wedge$, $\forall$. Note that $\sim$ is always applied in positive contexts and $\mathrel{\dot{=}}$ is always applied in negative contexts: the approximate equalities are reversed when negation is encountered.

The following theorem relates testing satisfaction to usual behavioural satisfaction ($\models_\approx$), where equality is interpreted as behavioural equality ($\approx$) and quantification is over all of $Dom(\approx)$.

Theorem 3.2 ([Mac99]). *If $\sim$ is complete, $\mathrel{\dot{=}}$ is sound, and ψ has only positive occurrences of $\forall$ and negative occurrences of $\exists$, then $A, \alpha \models_\approx \psi$ implies $A, \alpha \models^T_{\sim,\mathrel{\dot{=}}} \psi$.* □

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

4 Testing from Non-Generic Unit Specifications

This section discusses testing non-generic program units against specifications without considering the internal modular structure of the units. The styles of testing presented can be used to test the individual units of a modular system.

Good practice requires units to be checked independently of their contexts of use. For formal (functional) testing, this corresponds to checking whether the unit satisfies its specification. Testing from flat specifications can follow directly the approach presented in Sect. 3 for each test case. However, once structured specifications are considered, there are additional complications. First, the structure has to be taken into account when interpreting test results w.r.t. specification axioms. Also, in order to check axioms that involve "hidden symbols" it is necessary to provide an additional implementation for these symbols as the program under test is not required to implement them.

Structured specifications are built using structuring primitives like renaming, union, exporting and extension. These provide a powerful mechanism for reusing and adapting specification as requirements evolve. In *structured specifications with testing interface* [Mac00b], test sets are incorporated into specifications.

Definition 4.1 (Structured Specifications with Testing Interface). *The syntax and semantics of structured specifications are inductively defined as follows. Each specification SP is assigned a signature $Sig(SP)$ and two classes of $Sig(SP)$-algebras. $Mod_{\approx}(SP)$ is the class of* "real" models *of SP w.r.t. the family of Σ-behavioural equalities $\approx = (\approx_\Sigma)_{\Sigma \in Sign}$, and $ChMod_{\sim,\doteq}(SP)$ is the class of* "checkable" models *of SP determined by testing w.r.t. the families of approximate equalities $\sim = (\sim_\Sigma)_{\Sigma \in Sign}$ and $\doteq = (\doteq_\Sigma)_{\Sigma \in Sign}$ and the test sets associated with each axiom.*

1. (**Basic**) $SP = \langle \Sigma, \Psi \rangle$ *with* $\Psi \subseteq \{(\psi, T) \mid \psi \in Sen(\Sigma) \text{ and } T \subseteq T_\Sigma\}$.
   - $Sig(SP) = \Sigma$
   - $Mod_{\approx}(SP) = \{A \in Alg(\Sigma) \mid \bigwedge_{(\psi,T)\in\Psi} A \models_{\approx} \psi\}$
   - $ChMod_{\sim,\doteq}(SP) = \{A \in Alg(\Sigma) \mid \bigwedge_{(\psi,T)\in\Psi} A \models^T_{\sim,\doteq} \psi\}$
2. (**Union**) $SP = SP_1 \cup SP_2$, *where SP_1 and SP_2 are structured specifications, with $Sig(SP_1) = Sig(SP_2)$.*
   - $Sig(SP) = Sig(SP_1) \quad [= Sig(SP_2)]$
   - $Mod_{\approx}(SP) = Mod_{\approx}(SP_1) \cap Mod_{\approx}(SP_2)$
   - $ChMod_{\sim,\doteq}(SP) = ChMod_{\sim,\doteq}(SP_1) \cap ChMod_{\sim,\doteq}(SP_2)$
3. (**Renaming**) $SP = \mathsf{translate}\ SP'\ \mathsf{with}\ \sigma$, *where* $\sigma : Sig(SP') \to \Sigma$.
   - $Sig(SP) = \Sigma$
   - $Mod_{\approx}(SP) = \{A \in Alg(\Sigma) \mid A|_\sigma \in Mod_{\approx}(SP')\}$
   - $ChMod_{\sim,\doteq}(SP) = \{A \in Alg(\Sigma) \mid A|_\sigma \in ChMod_{\sim,\doteq}(SP')\}$
4. (**Exporting**) $SP = SP'|_\Sigma$, *where Σ is a subsignature of $Sig(SP')$.*
   - $Sig(SP) = \Sigma$
   - $Mod_{\approx}(SP) = \{A'|_\Sigma \mid A' \in Mod_{\approx}(SP')\}$
   - $ChMod_{\sim,\doteq}(SP) = \{A'|_\Sigma \mid A' \in ChMod_{\sim,\doteq}(SP')\}$

Operations presented in Definition 4.1 are primitive ones and, in practice, more complex operations, defined from their combination, are found in CASL and other languages. Extension – "**then**" in CASL – is defined in terms of renaming and union: $SP'\ \mathsf{then\ sorts}\ S\ \mathsf{opns}\ F\ \mathsf{axioms}\ \Psi \stackrel{\text{def}}{=} \langle \Sigma, \Psi \rangle \cup \mathsf{translate}\ SP'\ \mathsf{with}\ \sigma$, where SP' is a structured specification, S is a set of sorts, F is a set of function declarations, $\Sigma = Sig(SP') \cup (S, F)$, $\sigma : Sig(SP') \hookrightarrow \Sigma$ is the inclusion, and Ψ is a set of axioms over Σ with their associated test sets. The union of specifications over possibly different signatures – "**and**" in CASL – can be expressed as: $SP_1\ \mathsf{and}\ SP_2 \stackrel{\text{def}}{=} \mathsf{translate}\ SP_1\ \mathsf{with}\ \sigma_1 \cup \mathsf{translate}\ SP_2\ \mathsf{with}\ \sigma_2$, where $\Sigma = Sig(SP_1) \cup Sig(SP_2)$ and $\sigma_1 : Sig(SP_1) \hookrightarrow \Sigma$, $\sigma_2 : Sig(SP_2) \hookrightarrow \Sigma$.

To handle the oracle problem for structured specifications, two styles of testing are suggested in [Mac00b]: structured and flat testing. *Structured testing* of a Σ-algebra A against a structured specification SP corresponds to membership in the class of checkable models of SP, i.e., $A \in ChMod_{\sim,\doteq}(SP)$. Structured testing is based on the structure of SP; it may involve more than one set of test obligations (see Definition 4.1) and may demand additional implementation of symbols not in A (not exported by SP).

On the other hand, flat testing takes a monolithic approach based on an unstructured view of the specification without considering non-exported symbols

and using a single pair of approximate equalities on the overall signature of SP. More specifically, *flat testing* corresponds to testing satisfaction of axioms extracted from SP, i.e., $\bigwedge_{(\psi,T)\in TAx(SP)} A \models^T_{\sim_\Sigma,\doteq_\Sigma} \psi$ where $\Sigma = Sig(SP)$ and $TAx(SP)$ are the *visible axioms of* SP, defined as follows.

Definition 4.2 (Visible Axioms). *The set of* visible axioms *together with corresponding test sets of a specification SP is defined as follows.*

1. $TAx(\langle \Sigma, \Psi \rangle) = \Psi$
2. $TAx(SP_1 \cup SP_2) = TAx(SP_1) \ \cup \ TAx(SP_2)$
3. $TAx(\mathsf{translate}\ SP'\ \mathsf{with}\ \sigma) = \sigma(TAx(SP'))$
4. $TAx(SP'|_\Sigma) = \{(\phi, T \cap T_\Sigma) \ \mid \ (\phi, T) \in TAx(SP') \text{ and } \phi \in Sen(\Sigma)\}$

The visible axioms of a specification SP exclude those that refer to non-exported symbols, translating the rest to the signature of SP.

Whether structured or flat testing is performed, we must ask: under which conditions are correct models not rejected by testing? Results in [Mac00b,Mac00c] show that under certain assumptions, structured testing and flat testing do not reject correct models, even though incorrect ones can be accepted.

Theorem 4.3 ([Mac00b]). *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 $A \in ChMod_{\sim,\doteq}(SP)$.* □

Theorem 4.4 ([Mac00b]). *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 $\bigwedge_{(\psi,T)\in TAx(SP)} A \models^T_{\sim,\doteq} \psi$.* □

Structured testing is more flexible than flat testing in the sense that fewer assumptions are made. The family of observational equalities is not compatible as mentioned in Sect. 2, so the additional assumption is a strong one. On the other hand, flat testing is simpler. Both theorems cover a prevalent use of quantifiers. Their duals also hold, but are less interesting. There are also variants of these theorems that substitute assumptions on test sets for assumptions on quantifiers.

Structured testing and flat testing are two extremes. In practice, we may combine them. This can be done via normalization, where a structured specification SP is transformed into a specification $\boldsymbol{nf}(SP)$ of the form $\langle \Sigma', \Psi' \rangle|_\Sigma$ [BCH99] having the same signature and class of models. This procedure groups axioms, taking hidden symbols into account, so that the result is a flat specification which exports visible symbols. The usual normalization procedure is easily extended to structured specifications with testing interface (see [Mac00a] for details) and then we obtain the following result:

Theorem 4.5 ([Mac00a]). *If $\sim$ and $\doteq$ are compatible, then $ChMod_{\sim,\doteq}(SP) = ChMod_{\sim,\doteq}(\boldsymbol{nf}(SP))$.* □

The main advantage of normal form is to allow a combination of compositional and non-compositional testing, namely *semi-structured* testing, where

normal forms are used to replace *parts* of a specification, especially when these parts are combined by union. The result is to reduce the number of different experiments that are performed. Then, the resulting specification can be checked by structured testing and a result analogous to Theorems 4.3 and 4.4 is obtained:

Theorem 4.6 ([Mac00a]). *If $\sim$ is complete, $\hat{=}$ 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 $A \in ChMod_{\sim,\hat{=}}(\mathbf{nf}(SP))$.* □

Even though we have shown theoretical results regarding structured, flat and semi-structured testing, these styles of testing can be infeasible in practice when the structure of the program is not taken into account, since it may be necessary to decompose the program to reflect certain signatures in the structure of the specification and/or re-test the whole program every time it is modified.

5 Testing from Generic Unit Specifications

This section is concerned with testing generic units independent of particular instantiations. This is a difficult task for testing since we have to anticipate the behaviour of a generic unit when instantiated by specific units, but the set of all possible instantiations is almost always infinite. Not all units having the right signature are correct implementations of the argument specification, and correctness cannot generally be determined by testing. However, testing a generic unit using an incorrect implementation of the argument specification may lead to rejection of correct generic units.

The syntax and semantics of generic unit specifications are as follows. First, let $Alg(\Sigma' \to \Sigma) = \{F : Alg(\Sigma') \rightharpoonup Alg(\Sigma) \mid \forall A \in Dom(F), F[A]|_{\Sigma'} = A\}$ be the class of persistent partial functions taking Σ'-algebras to Σ-algebras, where $\Sigma' \subseteq \Sigma$. These functions on algebras model generic units; they are partial since a generic unit is only required to produce a result when instantiated by an algebra that satisfies the argument specification.

Definition 5.1 (Generic Unit Specifications). *Let $Sig(SP) = \Sigma$ and $Sig(SP') = \Sigma'$, such that SP extends SP', i.e. $\Sigma' \subseteq \Sigma$ and for all $A \in Mod_{\approx}(SP)$, $A|_{\Sigma'} \in Mod_{\approx}(SP')$.*

- $Sig(SP' \to SP) = \Sigma' \to \Sigma$
- $Mod_{\approx}(SP' \to SP) = \{F \in Alg(\Sigma' \to \Sigma) \mid \forall A \in Mod_{\approx}(SP'), A \in Dom(F)$ and $F[A] \in Mod_{\approx}(SP)\}$
- $ChMod_{\sim,\hat{=}}(SP' \to SP) = \{F \in Alg(\Sigma' \to \Sigma) \mid \forall A \in ChMod_{\sim,\hat{=}}(SP'), A \in Dom(F)$ and $F[A] \in ChMod_{\sim,\hat{=}}(SP)\}$

Let $SP' \to SP$ be a generic unit specification and let F be a generic unit that is claimed to correctly implement this specification. In contrast to testing from non-generic unit specifications, membership in the class of checkable models does not give rise to a feasible style of testing from generic unit specifications. First,

the class of models of SP' may be infinite. Moreover, the class of checkable models of $SP' \to SP$ cannot be directly compared to its class of "real" models. Suppose $F \in Mod_{\approx}(SP' \to SP)$ and $A \in ChMod_{\sim,\doteq}(SP')$, but $A \notin Mod_{\approx}(SP')$ and $F[A] \notin ChMod_{\sim,\doteq}(SP)$. Then, $F \notin ChMod_{\sim,\doteq}(SP' \to SP)$ – a correct F is rejected due to bugs in A.

As explained earlier, a testing method should ensure that correct models are not rejected. This requires that incorrect models of the parameter specification are not used in testing. Suppose another class of models, named *strong models*, is defined as an alternative to the class of checkable models.

Definition 5.2 (Strong Models). *The class of* strong models *of* $SP' \to SP$ *is defined as* $SMod_{\sim,\doteq}(SP' \to SP) = \{F \in Alg(\Sigma' \to \Sigma) \mid \forall A \in Mod_{\approx}(SP'), A \in Dom(F)$ and $F[A] \in ChMod_{\sim,\doteq}(SP)\}$.

This represents the class of models which are successfully tested when only correct implementations of SP' are considered. We then have:

Theorem 5.3. *If* $\sim$ *is complete,* $\doteq$ *is sound, and the axioms of* SP *have only positive occurrences of* $\forall$ *and negative occurrences of* $\exists$*, then* $F \in Mod_{\approx}(SP' \to SP)$ *implies* $F \in SMod_{\sim,\doteq}(SP' \to SP)$.

Proof. Suppose $F \in Mod_{\approx}(SP' \to SP)$. Then, $\forall A \in Mod_{\approx}(SP')$, $A \in Dom(F)$ and $F[A] \in Mod_{\approx}(SP)$. By Theorem 4.3, $F[A] \in ChMod_{\sim,\doteq}(SP)$. Hence, $F \in SMod_{\sim,\doteq}(SP' \to SP)$. □

This means that if we can test for membership in the class of strong models, then correct models of generic unit specifications are not rejected. Obviously, incorrect models can be accepted. As with Theorem 4.3, the dual also holds.

In practice, testing membership in $SMod_{\sim,\doteq}(SP' \to SP)$ (this also applies to $ChMod_{\sim,\doteq}(SP' \to SP)$) is not possible, since as already noted the class $Mod_{\approx}(SP')$ is almost always infinite and in any case membership in this class is not testable in general. A feasible approach to test whether generic units are models of $SP' \to SP$ should only rely on a finite subset of $Mod_{\approx}(SP')$. So let $\mathcal{C}$ be a set of units ("stubs") chosen according to some coverage criteria. Then we define a "weak" class of models of $SP' \to SP$ as follows.

Definition 5.4 (Weak Models). *Let* $\mathcal{C} \subseteq Mod_{\approx}(SP')$*. The class of* weak models *of* $SP' \to SP$ *is defined as* $WMod_{\sim,\doteq,\mathcal{C}}(SP' \to SP) = \{F \in Alg(\Sigma' \to \Sigma) \mid \forall A \in \mathcal{C},\ A \in Dom(F)$ *and* $F[A] \in ChMod_{\sim,\doteq}(SP)\}$.

The intention here is to select a class $\mathcal{C}$ which is finite and has a reasonable size such that F can be tested with this class and useful information gained. The following theorem shows that under certain assumptions, correct generic units are not rejected by testing w.r.t. the class of weak models.

Theorem 5.5. *If* $\sim$ *is complete,* $\doteq$ *is sound, and the axioms of* SP *have only positive occurrences of* $\forall$ *and negative occurrences of* $\exists$*, then* $F \in Mod_{\approx}(SP' \to SP)$ *implies* $F \in WMod_{\sim,\doteq,\mathcal{C}}(SP' \to SP)$ *for any* $\mathcal{C} \subseteq Mod_{\approx}(SP')$.

Proof. Suppose $F \in Mod_{\approx}(SP' \to SP)$. Then, $\forall A \in \mathcal{C} \subseteq Mod_{\approx}(SP')$, $A \in Dom(F)$ and $F[A] \in Mod_{\approx}(SP)$. By Theorem 4.3, $F[A] \in ChMod_{\sim,\doteq}(SP)$. Hence, $F \in WMod_{\sim,\doteq,\mathcal{C}}(SP' \to SP)$. □

The class of weak models is comparable to the classes of checkable and strong models, i.e., for any $\mathcal{C} \subseteq Mod_{\approx}(SP')$, $F \in ChMod_{\sim,\doteq}(SP' \to SP)$ implies $F \in WMod_{\sim,\doteq,\mathcal{C}}(SP' \to SP)$, provided the assumptions of Theorem 4.3 hold for SP', and $F \in SMod_{\sim,\doteq}(SP' \to SP)$ implies $F \in WMod_{\sim,\doteq,\mathcal{C}}(SP' \to SP)$. Moreover, for any algebra A used to test membership of F in the class of weak models of $SP' \to SP$, we can conclude that $F[A]$ is indeed a checkable model of SP, i.e., if $A \in \mathcal{C}$ and $F \in WMod_{\sim,\doteq,\mathcal{C}}(SP' \to SP)$ then $F[A] \in ChMod_{\sim,\doteq}(SP)$, by Definition 5.4. However, what if $A \in Mod_{\approx}(SP')$, but $A \notin \mathcal{C}$? Is $F[A] \in ChMod_{\sim,\doteq}(SP)$? How do we select an appropriate *finite* set of $Sig(SP')$-algebras so that an answer to the above question can be given?

Even though, under the assumptions of Theorem 5.5, testing membership in the class of weak models does not reject correct programs, not all sets $\mathcal{C}$ of stubs are equally interesting. It is desirable that a generic unit F be tested without regard to the units that are going to be used to instantiate it subsequently. Then, if we can conclude that $F[A]$ is a checkable model for some A, it may be possible to avoid re-testing F when A is replaced by a different unit. In other words, we need to select $\mathcal{C}$ to be a representative subset of $Mod_{\approx}(SP')$ so that given a correct realisation A of SP' ($A \in Mod_{\approx}(SP')$) and a generic unit F in the class of weak models of $SP' \to SP$ ($F \in WMod_{\sim,\doteq,\mathcal{C}}(SP' \to SP)$), we can conclude that $F[A]$ is a realisation of SP ($F[A] \in ChMod_{\sim,\doteq}(SP)$).

One possible answer to the above questions might be to pick one representative of every equivalence class w.r.t. an equivalence relation $\equiv$ on algebras when defining $\mathcal{C}$. This might be the observational equivalence on algebras [BHW95] or an approximation to it. The idea is similar to equivalence partitioning of test sets and the uniformity hypothesis in black-box testing [Ber91].

Let $\equiv$ be an equivalence relation and define the *closure of* $\mathcal{C}$ *under* $\equiv$ as $Cl_{\equiv}(\mathcal{C}) = \{A \in Alg(\Sigma) \mid \exists B \in \mathcal{C} \cdot A \equiv B\}$. In particular, we might pick $\mathcal{C}$ so that $Cl_{\equiv}(\mathcal{C})$ coincides with $Mod_{\approx}(SP)$. Following [BHW95], we focus on equivalence relations that are "factorizable" by partial congruences of interest. The idea comes from automata theory: two finite state machines are equivalent (i.e. accept the same language) iff quotienting each one by the Nerode equivalence on states yields isomorphic machines. (In fact, we will require only right factorizability.)

Definition 5.6 (Factorizability). *An equivalence* $\equiv \subseteq Alg(\Sigma) \times Alg(\Sigma)$ *is* factorizable *by a family of partial* Σ*-congruences* $\approx = (\approx_A)_{A \in Alg(\Sigma)}$ *if* $A \equiv B$ *iff* $A/\approx_A \cong B/\approx_B$*;* $\equiv$ *is* right factorizable *by* $\approx$ *if* $A \equiv B$ *implies* $A/\approx_A \cong B/\approx_B$*.*

It is shown in [BHW95] that various definitions of observational equivalence are factorizable by corresponding observational equalities. We will need an equivalence that is right factorizable by an approximate equality that is complete with respect to our chosen behavioural equality. Complete equalities are coarser than $\approx$, and if $\approx$ is observational equality then equivalences that are factorizable by such equalities are coarser than observational equivalence. Requiring

right factorizability permits the equivalence to be *finer* than the factorizable one, including observational equivalence and equivalences finer than that.

The following theorem relates behavioural satisfaction of a sentence and ordinary satisfaction of the same sentence in a quotient algebra.

Theorem 5.7 ([BHW95]). *Let $\approx = (\approx_A)_{A \in Alg(\Sigma)}$ be a family of partial Σ-congruences. Then $A/\approx_A \models \psi$ iff $A \models_\approx \psi$.* □

Putting these together:

Corollary 5.8. *Let $\equiv$ be right factorizable by $\approx$. Then $A \equiv B$ implies $A \models_\approx \psi$ iff $B \models_\approx \psi$.*

Proof. $A \models_\approx \psi$ iff $A/\approx_A \models \psi$ (Theorem 5.7) iff $B/\approx_B \models \psi$ ($A \equiv B$, right factorizability, preservation of satisfaction by $\cong$) iff $B \models_\approx \psi$ (Theorem 5.7). □

Theorem 5.9. *Let $\equiv$ be right factorizable by $\sim$. Then $A \equiv B$ implies $A \in ChMod_{\sim,\sim}(SP)$ iff $B \in ChMod_{\sim,\sim}(SP)$.*

Proof. By induction on the structure of SP, using Corollary 5.8 for specifications of the form $\langle \Sigma, \Psi \rangle$. Since test sets are finite sets of ground terms, $A \models^T_{\sim,\sim} \psi$ is equivalent to $A \models_\sim \psi'$ where ψ' is obtained from ψ by replacing each subformula of the form $\forall x : s \cdot \varphi$ by $\bigwedge_{t \in T_s} \varphi[t/x]$ and each subformula of the form $\exists x : s \cdot \varphi$ by $\bigvee_{t \in T_s} \varphi[t/x]$. □

A further assumption will be that generic units preserve $\equiv$, i.e. are "stable":

Definition 5.10 (Stability). *A generic unit $F \in Alg(\Sigma' \to \Sigma)$ is* stable *with respect to equivalences $\equiv_{\Sigma'} \subseteq Alg(\Sigma') \times Alg(\Sigma')$ and $\equiv_\Sigma \subseteq Alg(\Sigma) \times Alg(\Sigma)$ if for any $A \in Dom(F)$, $A \equiv_{\Sigma'} B$ implies $B \in Dom(F)$ and $F[A] \equiv_\Sigma F[B]$.*

Stability with respect to observational equivalence is a reasonable assumption for generic units expressed in a programming language, since stability is closely related to the security of the data encapsulation mechanisms in that language, see [Sch87] and [ST97]. For an equivalence that is only an approximation to observational equivalence, stability seems reasonable as a hypothesis in the context of testing. We then have the main result of this section:

Theorem 5.11. *Let $\mathcal{C} \subseteq Mod_\approx(SP')$. If $F \in WMod_{\sim,\sim,\mathcal{C}}(SP' \to SP)$, $A \in Cl_{\equiv_{\Sigma'}}(\mathcal{C})$, $\equiv$ is right factorizable by $\sim$ and F is stable with respect to $\equiv_{\Sigma'}$ and $\equiv_\Sigma$, then $F[A] \in ChMod_{\sim,\sim}(SP)$.*

Proof. $A \in Cl_{\equiv_{\Sigma'}}(\mathcal{C})$ means that $A \equiv_{\Sigma'} B$ for some $B \in \mathcal{C}$, and then $F[B] \in ChMod_{\sim,\sim}(SP)$. By stability, $F[A] \equiv_\Sigma F[B]$. Then, by Theorem 5.9, $F[A] \in ChMod_{\sim,\sim}(SP)$. □

Corollary 5.12. *Let $\mathcal{C} \subseteq Mod_\approx(SP')$. If $F \in WMod_{\sim,\sim,\mathcal{C}}(SP' \to SP)$, $\equiv$ is right factorizable by $\sim$ and F is stable with respect to $\equiv_{\Sigma'}$ and $\equiv_\Sigma$, then $F \in WMod_{\sim,\sim,Cl_\equiv(\mathcal{C})}(SP' \to SP)$.*

Theorem 5.11 and Corollary 5.12 are useful "amplification" results. They allow information gained from testing particular instantiations (the set of stubs $\mathcal{C}$) to be extrapolated to give information about instantiations that have not actually been tested. Theorem 5.11 relates to the practice of replacing a module in a working system (in this case, the parameter of a generic unit) with another version. If the two versions can be shown to be equivalent ($\equiv$), then the overall system will continue to work provided the assumptions in Theorem 5.11 are met. In Corollary 5.12, if we choose $\mathcal{C}$ so that $Cl_{\equiv}(\mathcal{C}) = Mod_{\approx}(SP)$, then the conclusion is equivalent to membership in the class of strong models of $SP' \to SP$. In most cases, this ideal will not be achievable; nevertheless, we can aim to include in $\mathcal{C}$ representatives of equivalence classes related to the particular class of applications for which F is intended to be used.

In order for testing to avoid rejecting correct units, according to Theorem 5.5 we need to restrict to specifications with only positive occurrences of $\forall$ and negative occurrences of $\exists$. A much more serious restriction when combining Theorems 5.5 and 5.11 is that $\sim$ needs to be both sound *and* complete. However, an analysis of the proof shows that it is sufficient if $\sim$ is complete for equations in positive positions and sound for equations in negative positions. If $\approx$ is observational equality then this can be achieved by taking $\sim$ to be a contextual equality (and hence complete) and restricting equations in negative positions to be of observable sorts only, for which $\sim$ is sound and complete.

6 Concluding remarks

We have presented ideas relating to testing modular systems against CASL-style architectural specifications. Our overall objective is to support independent development and verification of program components.

The problem of testing against architectural specifications reduces to:

1. testing non-generic units against structured specifications;
2. testing generic units against specifications of the form $SP' \to SP$; and
3. "integration testing" for unit terms that avoids re-testing.

Solutions to (1) are presented in Sect. 4, where previous results for testing against structured specifications are reviewed and discussed in the context of architectural specifications. Then, based on previous research on behavioural implementations, ideas concerning (2) are presented in Sect. 5. Since the class of possible parameter units ("stubs") is almost always infinite, we suggest that a representative finite class be selected, allowing testing results to be extrapolated to equivalent units. For this, stability of generic units is assumed. Problem (3) is future work, although it is already clear that the results in Sect. 5 and [Mac00b,Mac00a,Mac00c] are relevant.

Other questions for the future concern the circumstances under which our approximation to stability holds, as well as the connection between equivalence on algebras and testing satisfaction, including the question of how this equivalence can be effectively checked. We also aim to extend the results to specifications

of higher-order generic units. Finally, a general method of applying the ideas presented along with practical case studies are needed.

Acknowledgments: Thanks to Marie-Claude Gaudel for encouragement and to members of IFIP WG 1.3 for interesting questions. This research was partly funded by PROTEM-CC/CNPq.

References

[ABK+03] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. Mosses, D. Sannella and A. Tarlecki. CASL: The common algebraic specification language. *Theoretical Computer Science*. To appear (2003).

[BB01] H. Baumeister and D. Bert. Algebraic specification in CASL. In *Software Specification Methods – An Overview Using a Case Study*. Springer (2001).

[Ber89] G. Bernot. A formalism for test with oracle based on algebraic specifications. Report 89-4, LIENS/DMI, Ecole Normale Supérieure, Paris (1989).

[Ber91] G. Bernot. Testing against formal specifications: a theoretical view. *Proc. TAPSOFT'91*, Brighton. Springer LNCS 494, 99–119 (1991).

[BCH99] M. Bidoit, M.V. Cengarle and R. Hennicker. Proof systems for structured specifications and their refinements. *Algebraic Foundations of Systems Specifications*, chapter 11. Springer (1999).

[BH99] M. Bidoit and R. Hennicker. Observational logic. *Proc. AMAST'98*, Manaus. Springer LNCS 1548, 263–277 (1999).

[BHW95] M. Bidoit, R. Hennicker and M. Wirsing. Behavioural and abstractor specifications. *Science of Computer and Programming*, 25:149–186 (1995).

[BST02] M. Bidoit, D. Sannella and A. Tarlecki. Architectural specifications in CASL. *Formal Aspects of Computing*. To appear (2002).

[DW00] M. Doche and V. Wiels. Extended institutions for testing. *Proc. AMAST 2000*. Springer LNCS 1816, 514–528 (2000).

[Gau95] M.-C. Gaudel. Testing can be formal, too. *Proc. TAPSOFT'95*, Aarhus. Springer LNCS 915 (1995).

[LeG99] P. LeGall. *Vers une spécialisation des logiques pour spécifier formellement et pour tester des logiciels*. Habilitation thesis, Université d'Evry (1999).

[LA96] P. LeGall and A. Arnould. Formal specification and test: correctness and oracle. *Proc. WADT'95*, Oslo. Springer LNCS 1130 (1996).

[Mac99] P.D.L. Machado. On oracles for interpreting test results against algebraic specifications. *Proc. AMAST'98*, Manaus. Springer LNCS 1548, 502–518 (1999).

[Mac00a] P.D.L. Machado. The rôle of normalisation in testing from structured algebraic specifications. *Proc. WADT'99*, Bonas. Springer LNCS 1827, 459–476 (2000).

[Mac00b] P.D.L. Machado. Testing from structured algebraic specifications. *Proc. AMAST 2000*. Springer LNCS 1816, 529–544 (2000).

[Mac00c] P.D.L. Machado. *Testing from Structured Algebraic Specifications: The Oracle Problem*. PhD thesis, LFCS, University of Edinburgh (2000).

[ST97] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269 (1997).

[Sch87] O. Schoett. *Data Abstraction and the Correctness of Modular Programming*. PhD thesis, LFCS, University of Edinburgh (1987).