Testing Modular Systems Against CASL Architectural Specifications

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

[1] Systems and Computing Department, Federal University of Paraiba
patricia@dsc.ufpb.br
[2] Laboratory for Foundations of Computer Science, University of Edinburgh
dts@dcs.ed.ac.uk

Abstract. The problem of testing modular systems against algebraic specifications is discussed. In particular, 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. One problem in testing from the unit supplier's point of view is how to test units independently of the context of use. This is most acute for generic units where the particular instantiation cannot be predicted. On the other hand, users of units are concerned with the particular context of use – dictated by the architectural specification at hand – where one concern is how to take advantage of the testing that has already been done by the supplier. Ideas for tackling these problems are presented.

Keywords: Algebraic specification, specification-based testing, modular systems

1 Introduction

Improving the quality and reducing the production cost of software systems is of paramount importance. There is a current focus on developing systems composed of configurable modules with specified interfaces that encapsulate data and functionality – so-called *component-based systems*. Progress in this area requires effective ways of verifying such systems and their individual components.

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. This is a practical alternative to formal proof. Much work in this area has focused on theoretical and practical problems related to test case selection and (automatic) interpretation of test results [BGM91,DF93,Don97,GJ98]. Testing from algebraic specifications has been investigated, focusing on both "flat" specifications [Ber89,Gau95,LA96,Mac99] and structured specifications [LA96,Mac00b,DW00]. Testing is usually planned from concrete low-level specifications rather than from abstract ones, and 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 somehow 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 ways of interacting with the system, usually an infinite and impractical activity. When only finite sequences 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 time and effort on non-errors. The main goal of testing is to find errors – as many as possible – according to a previously-planned test coverage. Based on test results it is then possible to compute the defect error rate and forecast reliability. Accepting incorrect programs is not a major problem as long as coverage is adequate, since testing is not expected to guarantee correctness. Still, it is possible to make explicit the hypotheses that relate successful testing to correctness [Ber91], e.g. the expectation ("uniformity" hypothesis) that programs will produce similar outputs on similar inputs.

The aim of this paper is to address the problem of testing modular systems where parts are developed independently from CASL-style architectural specifications [ABK$^+$,BST], focusing on styles of testing that address the oracle problem and take advantage of tests already performed for individual units. In this context, the problem of testing reduces to the problem of testing units independently in such a way that their integration can be checked in a cost-effective way. In CASL, architectural specifications are used for describing the modular structure of software systems. In contrast, structure in ordinary CASL specifications is merely for presentation purposes. 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. In this paper, we focus on unit terms formed by instantiating generic units. This avoids complications introduced by other ways of forming unit terms while exposing most of the main issues. (Note that specifications of generic units are quite different from generic specifications in CASL and other languages: here it is the *unit* that is generic, not the *specification* [SST92].)

When testing modular systems and their parts, different perspectives may need to be considered. From a *supplier's* point of view, units have to be checked independently of the contexts in which they are going to be used. For formal testing, this corresponds to checking whether a unit satisfies its specification. 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. On the other hand, *users* of units are concerned with the particular context in which these units are used, dictated by the architectural specification at hand. They may assume units have already been verified. But, when one unit is replaced by another, what tests need to be performed? Under what circumstances is it possible to avoid full re-testing? These are the kinds of questions regarding integration testing that are addressed in this paper. Even though we focus on testing from algebraic specifications, problems discussed in this paper arise in other formal frameworks and solutions proposed here can possibly be adapted to them.

The paper is structured as follows. Preliminary definitions are presented in Section 2. We assume some familiarity with concepts of algebraic specification. Section 3 introduces formal testing from algebraic specifications through an example. Section 4 reviews previous results on testing non-generic units from structured specifications. These results are not new but they have not previously been considered in the context of architectural specifications. Section 5 presents the main results of this paper which concern testing of generic units. Then, based on results of Sections 4 and 5, Section 6 addresses testing modular systems from architectural specifications focusing on integration of components developed independently. Finally, we present some concluding remarks and pointers for further work.

2 Preliminary definitions

Algebraic Specifications. As a usual assumption, programs are modelled as algebras. A specification declares a set of symbols – the signature – and contains axioms giving required properties of these symbols. Structured specifications are formed from "flat" specifications using structuring operations like union, renaming, extension and export, see Section 4. Let $\Sigma = (S, F)$ be a *signature*[1] with $sorts(\Sigma) = S$ and $opns(\Sigma) = F$ and let $T_\Sigma(X)$ be the *Σ-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 Σ-terms t and t' of the same sort, $t = t'$ is a *Σ-equation*; *first-order Σ-formulas* are built from Σ-equations, logical connectives ($\neg$,$\wedge$,$\vee$,$\Rightarrow$,$\Leftrightarrow$) and quantifiers ($\forall$,$\exists$). Axioms in specifications are Σ-formulas without free variables, called *Σ-sentences*; the set of all Σ-sentences is written $Sen(\Sigma)$. A *Σ-algebra* A consists of an S-sorted set $|A|$, the *carrier sets*, and for each $f : s_1 \times \ldots \times s_n \to s \in \Sigma$, a function $f_A : |A|_{s_1} \times \ldots \times |A|_{s_n} \to |A|_s$. We restrict to algebras with non-empty carriers; the class of all such Σ-algebras is written $Alg(\Sigma)$. For any Σ-algebra A and *valuation* $\alpha : X \to |A|$, there is a unique Σ-homomorphism $\alpha^\# : T_\Sigma(X) \to A$ which extends α. The *value* of $t \in |T_\Sigma(X)|_s$ in A under α is then $\alpha^\#(t) \in |A|_s$. If $t \in T_\Sigma$, i.e., t is a *ground Σ-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 *σ-reduct* of a Σ-algebra A is the evident Σ'-algebra written $A|_\sigma$, or $A|_{\Sigma'}$ if σ is an inclusion.

Behavioural 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 data type. Therefore, it is not appropriate to simply

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

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 partial Σ-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, i.e., $\forall f : s_1 \dots s_n \longrightarrow s \in F$, $\forall a_i, b_i \in A_{s_i}$, if $a_i \approx_{A,s_i} b_i$ for all $1 \le i \le n$, then $f_A(a_1,\dots,a_n) \approx_{A,s} f_A(b_1,\dots,b_n)$. The domain of definition of $\approx_A$ is $Dom(\approx_A) = \{a \mid a \approx_A a\}$. Let $Obs \subseteq S$ be a distinguished set of observable sorts. The partial *observational equality* $\approx_{Obs,A} = (\approx_{Obs,A,s})_{s\in S}$ is one example of a behavioural equality, where related elements are those that cannot be distinguished by observable computations.[2] A *Σ-behavioural equality* is defined as $\approx_\Sigma = (\approx_{\Sigma,A})_{A\in Alg(\Sigma)}$, one behavioural equality for each Σ-algebra A. Whenever Σ is obvious, $\approx$ is used without subscript to denote $\approx_\Sigma$. When A is also obvious, $\approx$ and $\approx_\Sigma$ are used to denote $\approx_{\Sigma,A}$.

Approximate Equality. 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* [Mac99,Mac00c] which are binary relations on values of the algebra. When compared to a behavioural equality, an approximate equality is *sound* if all values that it identifies are indeed equal, or *complete* if all equal values are identified (*sound* $\subseteq$ *behavioural* $\subseteq$ *complete*). A *contextual equality* $\sim_{\mathcal{C},A}$ is defined from a subset $\mathcal{C}$ of the observable computations.[3] 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.

Families of Equalities. The signature of different parts of a structured specification may be different, and the interpretation of equality (for example observational equality) may depend on the signature. Therefore, if we aim to deal with structured specifications, we need to consider *families* of equalities indexed by signatures. In the sequel, let $\approx = (\approx_\Sigma)_{\Sigma\in Sign}$ be a family of behavioural equalities, one for each signature Σ, where $Sign$ is the category of signatures. Likewise, let $\sim = (\sim_\Sigma)_{\Sigma\in Sign}$ and $\simeq = (\simeq_\Sigma)_{\Sigma\in Sign}$ denote families of approximate equalities. The family $\sim$ is complete (sound) w.r.t. $\approx$ iff $\forall \Sigma \in Sign$, $\sim_\Sigma$ is complete (sound) w.r.t. $\approx_\Sigma$. The *reduct* of a Σ-approximate equality $\sim_\Sigma$ by the morphism $\sigma : \Sigma' \to \Sigma$ considers only the relations of sorts mapped from Σ', i.e., $(\sim_\Sigma)|_\sigma = ((\sim_{\Sigma,A})|_\sigma)_{A\in Alg(\Sigma)}$ where $(\sim_{\Sigma,A})|_\sigma = ((\sim_{\Sigma,A})_{\sigma(s)})_{s\in S'}$. The family $\sim$ is *compatible* with signature morphisms if for all $\sigma : \Sigma' \to \Sigma$ and all Σ-algebras A, $\sim_{\Sigma',A|_\sigma} = (\sim_{\Sigma,A})|_\sigma$. For the results below that have compatibility

[2] 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 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 $\alpha_b(z_s) = b$.

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

as a condition, it is sufficient if equality on a given sort coincides for all signatures arising in the structure of a given specification. Compatibility may fail if these signatures have different sets of observers for the same sort. The family of literal set-theoretical equalities $=$ on values of an algebra is always compatible. However, it is easy to check that the family of observational equalities is not compatible. Suppose $\sigma : \Sigma' \hookrightarrow \Sigma$ is an inclusion. In this case, Σ may have more observers than Σ'. Thus, $(\approx_{\Sigma,\sigma(Obs')})|_\sigma$ may be finer than $\approx_{\Sigma',Obs'}$. One may restrict the introduction of new observers to avoid this problem, see e.g. [BH99].

3 Formal Testing from Algebraic Specifications

This section presents formal testing from algebraic specifications through an example and based on a generic model of the testing process. For simplicity, we assume programs to be tested are written in SML, viewing structures as algebras and using boolean-valued functions to model predicates. As mentioned before, we focus on test oracles.

Testing from algebraic specifications boils down to checking whether specification axioms are satisfied by programs [Gau95]. 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. The generic testing model to be followed consists of the following activities: test case selection, test data selection, test oracle design, test execution and interpretation of results. *Test cases* are extracted from specifications together with *test sets* which are defined at specification level and associated with axioms. Then, oracles are defined for each test case or group of test cases. Oracles are predicates to evaluate test cases according to test results incorporating procedures to compute equality on non-observable sorts. A *test obligation* corresponds to the combination of a test case, test data, a test oracle and a program to be tested. Whenever there is no chance of confusion, this is referred to simply as a *test*. In the sequel, a simple example is used to illustrate these ideas.

Example 3.1 (The INVOICE *Specification).* Throughout this paper we use the specification of “invoice orders” by Baumeister and Bert [BB01]. For the sake of space, we focus on a simplified version of the INVOICE specification. The main data involved in the problem domain are orders, stocks and products. The ORDER specification below defines orders. This specification imports the CASL basic specification of natural numbers, NAT. The sort *Nat*, its subsort *Pos* (the positive natural numbers) and the booleans (used when predicates are represented as operations) will be the observable sorts.

spec ORDER = NAT **then**
  sorts *Order*, *Product*;
  ops $ref : Order \to Product$;
    $ordered\_qty : Order \to Pos$;
    $mk\_order : Product * Pos \to Order$;
  preds $is\_pending, is\_invoiced : Order$;
  var $o : Order \bullet \neg is\_pending(o) \Leftrightarrow is\_invoiced(o)$ (1)

end

Each order references a single product given by the *ref* operation and the quantity of the ordered product, given by *ordered_qty*, is greater than zero. The state of an order is either "pending" or "invoiced" (Axiom 1). Note that axioms are implicitly universally quantified by all the declared variables. The specification also introduces the *Product* sort. The STOCK specification below defines stocks.

spec STOCK = NAT **then**
  sorts *Stock*, *Product*;
  ops $qty : Product \times Stock \to? Nat$;
    $add, remove : Product \times Pos \times Stock \to? Stock$;
  pred $\_\_isIn\_\_ : Product \times Stock$;
  vars $p, k : Product$; $n : Pos$; $s : Stock$
- $(def\ qty(p, s) \Leftrightarrow p\ isIn\ s) \wedge (def\ add(p, n, s) \Leftrightarrow p\ isIn\ s)$ (2)
- $def\ remove(p, n, s) \Leftrightarrow p\ isIn\ s \wedge qty(p, s) \geq n$ (3)
- $qty(p, add(p, n, s)) = qty(p, s) + n\ if\ p\ isIn\ s$ (4)
- $qty(k, add(p, n, s)) = qty(k, s)\ if\ p\ isIn\ s \wedge k\ isIn\ s \wedge k \neq p$ (5)
- $qty(p, remove(p, n, s)) = qty(p, s) - n\ if\ p\ isIn\ s \wedge qty(p, s) \geq n$ (6)
- $qty(k, remove(p, n, s)) = qty(k, s)\ if\ p\ isIn\ s \wedge k\ isIn\ s \wedge k \neq p$ (7)

end

Here, *qty* gives the quantity of a product in stock, *add* and *remove* update the quantity of a product in stock, and *isIn* returns whether or not a product is in stock. The *qty*, *add* and *remove* operations are only defined for products that are in stock. Also, *remove* can be applied only if there are enough items of the product (Axiom 3). An observational axiomatisation of *add* and *remove* is given based on their effect on the result of *qty* (Axioms 4 to 7). The INVOICE specification is given below. This extends the sum of ORDER and STOCK.

spec INVOICE = ORDER **and** STOCK **then**
  free type $OrdStk ::= mk(order\_of : Order; stock\_of : Stock)$;
  pred $referenced(o : Order; s : Stock) \Leftrightarrow ref(o)\ isIn\ s$; (8)
  pred $enough\_qty(o : Order; s : Stock) \Leftrightarrow ordered\_qty(o) \leq qty(ref(o), s)$; (9)
  pred $invoice\_ok(o : Order; s : Stock) \Leftrightarrow$
    $is\_pending(o) \wedge referenced(o, s) \wedge enough\_qty(o, s)$; (10)
  op $invoice\_order : Order \times Stock \to OrdStk$;
  vars $o : Order$; $s : Stock$
- $is\_invoiced(order\_of(invoice\_order(o, s)))\ if\ invoice\_ok(o, s)$ (11)
- $stock\_of(invoice\_order(o, s)) = remove(ref(o), ordered\_qty(o), s)$
    $if\ invoice\_ok(o, s)$ (12)
- $order\_of(invoice\_order(o, s)) = o\ if\ \neg invoice\_ok(o, s)$ (13)
- $stock\_of(invoice\_order(o, s)) = s\ if\ \neg invoice\_ok(o, s)$ (14)
- $ref(order\_of(invoice\_order(o, s))) = ref(o)$ (15)
- $ordered\_qty(order\_of(invoice\_order(o, s))) = ordered\_qty(o)$ (16)

hide $referenced, enough\_qty$
end

The *invoice_order* operation gets an order and a stock and returns a new order and stock where the state of the order is changed to invoiced (Axiom 11) and the quantity of the ordered product in stock is reduced by the ordered quantity (Axiom 12). The conditions required to invoice an order (*invoice_ok*) are: (1)

the state of the order is "pending" (*is_pending*), (2) the ordered product is in stock (*referenced*), (3) the ordered quantity is less than or equal to the quantity in stock (*enough_qty*). When these conditions are not fulfilled, the order and the stock are not modified (Axioms 13 and 14). The other attributes of the order are not changed by the *invoice_order* operation (Axioms 15 and 16). □

Test Case Selection. For simplicity, each axiom is regarded as a separate test case. Obviously, techniques can always be applied to simplify test cases or make them more practical [Don97]. But this out of the scope of this paper. In the sequel, we consider Axiom 14 as a test case. Given an implementation of Invoice, named `ImpInvoice`, we define test data and an oracle for checking whether this implementation satisfies the axiom.

Test Data Selection. Test 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. Moreover, this makes it possible to design tests as specifications are created. Test sets are defined here as sets of ground terms [Gau95,Mac99] which correspond to sets of values in the program under test.

Example 3.2 (Test Data Set). In order to check Axiom 14, two test sets are required: a set of orders and a set of stocks. These sets can be defined as follows: $T_{14,Order} = \{mk\_order(p_1, 4), mk\_order(p_1, 1), mk\_order(p_2, 8)\}$ and $T_{14,Stock} = \{add(p_1, 5, s_1), add(p_1, 2, add(p_2, 10, s_1))\}$ where p_1, p_2 are products and s_1 is a stock so that $p_1 \ isIn \ s_1$ and $p_2 \ isIn \ s_1$. □

It is crucial to apply an appropriate test set selection technique when defining test sets. Test sets shown above are used for illustrative purposes only. Automatic test set selection can be based on deterministic selection [Mar91] and/or probabilistic selection [BBL97].

Test Oracle Design The oracle problem for flat specifications often reduces to the problem of comparing two values of a non-observable sort for equality; when equality is to be interpreted as behavioural equality, for instance observational equality, it may be difficult or impossible to decide. Also, the use of universal and existential quantifiers ranging over infinite domains can make the oracle problem more difficult. An approach to defining oracles that addresses these problems is presented in [Mac99], where equality on non-observable sorts 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[4]. To handle the quantifier problem, restrictions are placed on the syntactic contexts in which they can occur. An *approximate oracle* is then a procedure that decides whether certain specification axioms are 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

[4] A context is *positive* if it is formed by an even number of applications of negation (e.g. ϕ is positive in both $\phi \wedge \psi$ and $\neg\neg\phi$). Otherwise, the context is *negative.* Note that ϕ is negative and ψ is positive 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.

quantifiers range only over given test sets. Note that, in this paper, oracles are also test drivers, i.e., they are also responsible for conducting the necessary tests.

Definition 3.3 (Testing Satisfaction). *Let Σ be a signature, $T \subseteq T_\Sigma$ be a Σ-test set and $\sim, \simeq$ be two Σ-approximate equalities. Let A be a Σ-algebra and $\alpha : X \to Dom(\approx_A)$ be a valuation. The* testing satisfaction relation *denoted by $\models^T_{\sim,\simeq}$ is defined as follows.*

1. *$A, \alpha \models^T_{\sim,\simeq} t = t'$ iff $\alpha^\#(t) \sim_A \alpha^\#(t')$;*
2. *$A, \alpha \models^T_{\sim,\simeq} \neg\psi$ iff $A, \alpha \models^T_{\simeq,\sim} \psi$ does not hold;*
3. *$A, \alpha \models^T_{\sim,\simeq} \psi_1 \wedge \psi_2$ iff both $A, \alpha \models^T_{\sim,\simeq} \psi_1$ and $A, \alpha \models^T_{\sim,\simeq} \psi_2$ hold;*
4. *$A, \alpha \models^T_{\sim,\simeq} \forall x{:}s \cdot \psi$ iff $A, \alpha[x \mapsto v] \models^T_{\sim,\simeq} \psi$ 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 $\simeq$ is always applied in negative contexts. Note that 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 3.4 ([Mac99]). *If $\sim$ is complete, $\simeq$ is sound, and ψ has only positive occurrences of $\forall$ and negative occurrences of $\exists$, then $A, \alpha \models_\approx \psi$ implies $A, \alpha \models^T_{\sim,\simeq} \psi$.* $\square$

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.

Example 3.5 (Approximate Oracle). An oracle for checking axiom 14 can be defined in SML as a boolean function that evaluates the axiom according to a test set and an implementation of Invoice.

```
fun oracle_14 (lo,ls) = forall lo (fn o => forall ls (fn s =>
  stock_of(invoice_order(o,s)) == s if not invoice_ok(o,s)))
```

where lo, ls are test sets of *Order* and *Stock* defined as in Example 3.2 and `forall : 'a list -> ('a -> bool) -> bool` is an implementation of the $\forall$ quantifier. Note that an implementation of `== : Stock * Stock -> bool` is required in order to make the oracle function executable. According to Theorem 3.4, `==` must be complete because it occurs in a positive context. (If we were checking axiom 5 or 7, we would instead need a sound implementation of equality on `Product` since there it occurs in a negative context.) The following is an implementation of `==` as a contextual equality:

```
fun s==s' (lp) =
  forall lp (fn p => (p isIn s) = (p isIn s') andalso qty(p,s) = qty(p,s'))
```

where lp is a list of products. If lp is appropriately chosen, this equality will be a good approximation to the real one. $\square$

Test Execution and Interpretation of Results. Given the `ImpInvoice` implementation, the `oracle_14` function can be run to execute the necessary tests: `oracle_14(`$T_{14,Order}$, $T_{14,Stock}$`)` exercises the operations of `ImpInvoice` to check satisfaction of Axiom 14. According to Theorem 3.4, if the result is `false` (i.e. `ImpInvoice` $\not\models^{T_{14}}_{==,\doteqdot}$ axiom 14 for any $\doteqdot$), then one of the exercised functions has a bug (i.e. `ImpInvoice` $\not\models_{\approx}$ axiom 14). If the result is `true`, we cannot conclude that `ImpInvoice` is correctly implemented. However, depending on the test set selection technique used, it may be possible to compute the degree of confidence that the program is correct – see [Mar91,BBL97] for details.

4 Testing from Non-Generic Unit Specifications

This section is concerned with testing non-generic program units against specifications without considering any internal modular structure the units may possess. In other words, units are viewed as monolithic modules. 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 the contexts in which they are going to be used. 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 Section 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 such as *referenced* and *enough_qty* in INVOICE, 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" (correct) 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 determined by testing w.r.t. the families of approximate equalities $\sim = (\sim_\Sigma)_{\Sigma \in Sign}$ and $\doteqdot = (\doteqdot_\Sigma)_{\Sigma \in Sign}$ and the test sets associated with each axiom.*

1. (**Basic**) $SP = \langle \Sigma, \Psi \rangle$ *with* $\Psi \subseteq \{(\psi, T) \mid \psi \in Sen(\Sigma) \text{ and } T \subseteq T_\Sigma\}$.
   – $Sig(SP) = \Sigma$
   – $Mod_{\approx}(SP) = \{A \in Alg(\Sigma) \mid \bigwedge_{(\psi,T)\in\Psi} A \models_{\approx} \psi\}$
   – $ChMod_{\sim,\doteqdot}(SP) = \{A \in Alg(\Sigma) \mid \bigwedge_{(\psi,T)\in\Psi} A \models^{T}_{\sim,\doteqdot} \psi\}$
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 – can be defined in terms of renaming and union: $SP'\ \mathsf{then\ sorts}\ S\ \mathsf{opns}\ F\ \mathsf{axioms}\ \Psi \overset{\text{def}}{=} \langle \Sigma, \Psi \rangle \cup \mathsf{translate}\ SP'\ \mathsf{with}\ \sigma$, 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: $SP_1\ \mathsf{and}\ SP_2 \overset{\text{def}}{=} \mathsf{translate}\ SP_1\ \mathsf{with}\ \sigma_1 \ \cup\ \mathsf{translate}\ SP_2\ \mathsf{with}\ \sigma_2$, where $\Sigma = Sig(SP_1) \cup Sig(SP_2)$ and $\sigma_1 : Sig(SP_1) \hookrightarrow \Sigma$, $\sigma_2 : Sig(SP_2) \hookrightarrow \Sigma$.

Example 4.2 (Structured Specifications). In Example 3.1, augmenting INVOICE by attaching a test set to each axiom would give a structured specification with testing interface that extends the sum of ORDER and STOCK. Note that *referenced* and *enough_qty* are auxiliary symbols, introduced to help express the properties of the remaining symbols, that are not exported by INVOICE. Therefore, they are not necessarily implemented in models of this specification. □

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., $A \in ChMod_{\sim,\doteq}(SP)$, whereas *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 4.3 (Visible Axioms). *The set of* visible axioms *together with corresponding test sets of a specification* SP *can be 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.

Example 4.4 (Visible axioms). TAx(INVOICE) consists of all the axioms of INVOICE (including those inherited from ORDER and STOCK) except for axioms 8, 9 and 10 as *referenced* and *enough_qty* are not exported. Any terms in the test sets associated with other axioms that refer to either of these two operations will not be included in the result. □

Structured testing is based on the structure of SP; it may consist of 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). Flat testing is a monolithic experiment 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.

Example 4.5 (Structured and Flat Testing). Structured testing of `ImpInvoice` against INVOICE incurs the following test obligations (after some simplifications). Let INVOICE-H be the INVOICE specification without **hide** and let `IH` be an implementation of *referenced* and *enough_qty*.

$$\begin{array}{l} \bigwedge_{(\psi,T)\in\ \Psi_I} \texttt{ImpInvoice} + \texttt{IH} \models^T_{\sim_{\Sigma I},\simeq_{\Sigma I}} \psi \\ \bigwedge_{(\psi,T)\in\ \Psi_S} \texttt{ImpInvoice}|_{\Sigma S} \models^T_{\sim_{\Sigma S},\simeq_{\Sigma S}} \psi \\ \bigwedge_{(\psi,T)\in\ \Psi_O} \texttt{ImpInvoice}|_{\Sigma O} \models^T_{\sim_{\Sigma O},\simeq_{\Sigma O}} \psi \end{array}$$

where Ψ_I, Ψ_S, Ψ_O are the sets of axioms in INVOICE-H, STOCK and ORDER; ΣI, ΣS, ΣO are the signatures of INVOICE-H, STOCK and ORDER; and $\sim$, $\simeq$ are families of approximate equalities. Note that different equalities on the same sort may be used for testing axioms in different specifications. On the other hand, flat testing incurs the following single obligation:

$$\bigwedge_{(\psi,T)\in\ TAx(\text{INVOICE})} \texttt{ImpInvoice} \models^T_{\sim_\Sigma,\simeq_\Sigma} \psi$$

where Σ is the signature of INVOICE. □

Whether structured or flat testing is performed, the following must be considered: under which conditions are correct models not rejected by testing? Results presented 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. These results are a generalization of Theorem 3.4.

Theorem 4.6 ([Mac00b]). *If $\sim$ is complete, $\simeq$ 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 $A \in ChMod_{\sim,\simeq}(SP)$.* □

Theorem 4.7 ([Mac00b]). *If $\sim$ is complete and compatible and $\simeq$ is sound and compatible and the axioms of SP have only positive occurrences of $\forall$ and negative occurrences of $\exists$, then $A \in Mod_\approx(SP)$ implies $\bigwedge_{(\psi,T)\in TAx(SP)} A \models^T_{\sim,\simeq} \psi$.* □

(These results have fewer assumptions than those in [Mac00b], owing to the association of test sets with individual axioms rather than with specifications there.)

Example 4.8 (Structured and Flat Testing, continued). The specifications in Example 4.5 fit the requirements of Theorems 4.6 and 4.7, but $\sim_{\Sigma I}$, $\sim_{\Sigma S}$ and $\sim_{\Sigma O}$ must be complete and $\doteq_{\Sigma I}$, $\doteq_{\Sigma S}$ and $\doteq_{\Sigma O}$ must be sound. We can define $\sim_{\Sigma O}$ on *Order* as a contextual equality, with a related definition of $\sim_{\Sigma I}$:

$$\begin{array}{lll} o \sim_{\Sigma O,A,Order} o' & \text{iff} & ordered\_qty_A(o) = ordered\_qty_A(o') \wedge \\ & & is\_pending_A(o) = is\_pending_A(o') \wedge \\ & & is\_invoiced_A(o) = is\_invoiced_A(o') \\ o \sim_{\Sigma I,A,Order} o' & \text{iff} & o \sim_{\Sigma O,A,Order} o' \wedge \\ & & \forall s \in |A|_{Stock} \cdot invoice\_OK_A(o,s) = invoice\_OK_A(o',s) \end{array}$$

and $\doteq$ can be defined based on comparing values of the concrete representation. For flat testing, both families $\sim$ and $\doteq$ must also be compatible, requiring $\sim_{\Sigma I}$, $\sim_{\Sigma S}$, $\sim_{\Sigma O}$ and $\sim_{\Sigma}$ to coincide on the sorts they have in common. But this does not hold: $\sim_{\Sigma I,A,Order}$ will in general be finer than $\sim_{\Sigma O,A|_{\Sigma O},Order}$. Neither distinguishes between orders of different products, but due to the definition of *invoice_OK* (axiom 10), $\sim_{\Sigma I,A,Order}$ can discriminate between orders of products that are in stock and not in stock. Therefore, a definition of $\sim_{\Sigma I,A,Order}$ that avoids use of *invoice_OK* should be used if compatibility is required. □

Clearly structured testing is more flexible than flat testing in the sense that fewer assumptions are made. As mentioned in Section 2, the family of observational equalities is not compatible, 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 variations 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 take advantage of their combination. One way to do this is via normalization, where a structured specification SP is transformed into an equivalent specification $\boldsymbol{nf}(SP)$ of the form $\langle \Sigma', \Psi' \rangle|_{\Sigma}$ [BCH99]. The intention is to handle the complexity of structured specifications by grouping axioms, taking hidden symbols into account, so that the result is a flat specification which exports visible symbols. The usual normalization procedure is easily extended to structured specifications with testing interface (see the appendix for the details) and then we obtain the following result:

Theorem 4.9 ([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 group definitions and, consequently, to reduce the number of different experiments that need to be performed. Then, the resulting specification can be checked by structured testing and a result analogous to Theorems 4.6 and 4.7 can be obtained:

Theorem 4.10 ([Mac00a]). *If* $\sim$ *is complete,* $\doteq$ *is sound,* $\approx$ *is compatible and the axioms of* SP *have only positive occurrences of* $\forall$ *and negative occurrences of* $\exists$, *then* $A \in Mod_{\approx}(SP)$ *implies* $A \in ChMod_{\sim,\doteq}(\boldsymbol{nf}(SP))$. □

Example 4.11 (Semi-Structured Testing). In Example 4.5, Invoice can be replaced by its normal form,[5] and then structured testing gives the following test obligation:

$$\bigwedge_{(\psi,T)\in\ \Psi_O\cup\Psi_S\cup\Psi_I} \mathtt{ImpInvoice} + \mathtt{IH} \models^{T}_{\sim_{\Sigma I},\hat{=}_{\Sigma I}} \psi$$

Here, the same axioms are tested as in structured testing against the original specification, but as a single combined obligation. The approximate equality $\sim_{\Sigma I}$ can be more accurate than $\sim_{\Sigma S}$ and $\sim_{\Sigma O}$ since it can be based on all the observers in Invoice-H, giving more accurate test results. Note that $\approx$ is not compatible – see Example 4.8. But if the sort *Product* is regarded as observable then $\approx$ *is* compatible for the signatures arising in Invoice, and so in that case the assumptions on Theorem 4.10 are met. □

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 ($\mathtt{ImpInvoice}|_{\Sigma OS}$) and/or re-test the whole program every time a single modification is introduced. Clearly, a good approach to testing modular programs should take the benefits of structured testing and normalisation into account without requiring unnatural decomposition of programs and allow for independent development and verification of parts of the program. Architectural specifications play a role here, and this is further discussed in Section 6. But before that, we need to look into how a *generic* unit can be tested independently of actual units used to instantiate it. This is the subject of the following section.

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 functions taking Σ'-algebras to Σ-algebras, where $\Sigma' \subseteq \Sigma$.

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 $A \in Mod_{\approx}(SP)$, $A|_{\Sigma'} \in Mod_{\approx}(SP')$.*

[5] This gives the same overall result as replacing just Invoice-H by its normal form, which justifies the title of the example.

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

Example 5.2 (Generic Unit Specification). A generic unit *OrderFun* : Nat $\to$ Order can be defined to give a realisation of Order when given a realisation of Nat. Genericity arises here to allow independent development of *OrderFun* and the chosen implementation of Nat. Of course, an implementation of Nat can be imported from a library rather than being developed from scratch. $\square$

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 5.3 (Strong Models). *The class of* strong models *of $SP' \to SP$ is defined as $SMod_{\sim,\doteq}(SP' \to SP) = \{F \in Alg(\Sigma' \to \Sigma) \mid \forall A \in Mod_{\approx}(SP'),$ $F[A]$ is defined 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 5.4. *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, $\forall A \in Mod_{\approx}(SP')$, $F[A]$ is defined and $F[A] \in Mod_{\approx}(SP)$. By Theorem 4.6, $F[A] \in ChMod_{\sim,\doteq}(SP)$. Hence, $F \in SMod_{\sim,\doteq}(SP' \to SP)$. $\square$

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 4.6, the dual also holds, but is less interesting.

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 5.5 (Weak Models). *Let* $\mathcal{C} \subseteq Mod_{\approx}(SP')$. *The class of* weak models *of* $SP' \to SP$ *is defined as* $WMod_{\sim,\doteq,\mathcal{C}}(SP' \to SP) = \{F \in Alg(\Sigma' \to \Sigma) \mid \forall A \in \mathcal{C},\ F[A]$ is defined 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 5.6. *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, $\forall A \in \mathcal{C} \subseteq Mod_{\approx}(SP')$, $F[A]$ is defined and $F[A] \in Mod_{\approx}(SP)$. By Theorem 4.6, $F[A] \in ChMod_{\sim,\doteq}(SP)$. Hence, $F \in WMod_{\sim,\doteq,\mathcal{C}}(SP' \to SP)$. □

Example 5.7 (Weak Models). Suppose we want to test whether *OrderFun* from Example 5.2 is a weak model of NAT $\to$ ORDER. For this, we need to select an appropriate class $\mathcal{C}$ of implementations of NAT. Then, the following test obligation, after some simplifications, is incurred:

$$\forall N \in \mathcal{C} \cdot \bigwedge_{(\psi,T)\in\Psi_O} OrderFun[N] \models^{T}_{\sim,\doteq} \psi$$

where Ψ_O are the axioms in ORDER. According to Theorem 5.6, if *OrderFun* is correct then this will hold. □

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 4.6 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 5.5. 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 5.6, 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 $\mathcal{C}$ as 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 ($F[A] \in ChMod_{\sim,\doteq}(SP)$). Moreover, even though we are only considering correct realisations to be included in $\mathcal{C}$, we might want to consider the case where A is a checkable model, but not necessarily a real model. In other words, can we select a representative subset $\mathcal{C}$ of $Mod_{\approx}(SP')$ so that $A \in ChMod_{\sim,\doteq}(SP')$ and $F \in WMod_{\sim,\doteq,\mathcal{C}}(SP' \to SP)$ implies $F[A] \in ChMod_{\sim,\doteq}(SP)$?

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 $Mod_{\approx}(SP)$ or $ChMod_{\sim,\doteq}(SP)$. Following [BHW95], we focus on equivalence relations that are "factorizable" by partial congruences of interest. (In fact, we will require only right factorizability.)

Definition 5.8 (Factorizability). *An equivalence relation* $\equiv\, \subseteq Alg(\Sigma) \times Alg(\Sigma)$ *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 be interested in equivalences that are right factorizable by an approximate equality that is complete with respect to our chosen notion of behavioural equality. Complete equalities are coarser than $\approx$, and equivalences that are factorizable by such equalities are coarser than observational equivalence. But requiring *right* factorizability permits the equivalence to be *finer* than the factorizable one, including observational equivalence and equivalences finer than that.

The following theorem gives a fundamental relationship between behavioural satisfaction of a sentence and ordinary satisfaction of the same sentence in a quotient algebra.

Theorem 5.9 ([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 5.10. *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 5.9) iff $B/{\approx_B} \models \psi$ ($A \equiv B$, right factorizability, preservation of satisfaction by $\cong$) iff $B \models_{\approx} \psi$ (Theorem 5.9). □

Theorem 5.11. *Let* $\equiv$ *be* right factorizable *by* $\sim$*. If the axioms of* SP *have equations in positive positions only, then* $A \equiv B$ *implies* $A \in ChMod_{\sim,\doteq}(SP)$ *iff* $B \in ChMod_{\sim,\doteq}(SP)$*.*

Proof. By induction on the structure of SP, using Corollary 5.10 for specifications of the form $\langle \Sigma, \Psi \rangle$. Since equations are in positive positions only and test sets are finite sets of ground terms, $A \models^{T}_{\sim,\doteq} \psi$ 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 5.12 (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 5.13. *If $F \in WMod_{\sim,\doteq,\mathcal{C}}(SP' \to SP)$, $A \in Cl_{\equiv_{\Sigma'}}(\mathcal{C})$, $\equiv_{\Sigma}$ is* right factorizable *by $\sim_{\Sigma}$, the axioms of SP have equations in positive positions only and F is stable with respect to $\equiv_{\Sigma'}$ and $\equiv_{\Sigma}$, then $F[A] \in ChMod_{\sim,\doteq}(SP)$.*

Proof. $A \in Cl_{\equiv_{\Sigma'}}(\mathcal{C})$ means that $A \equiv_{\Sigma'} B$ for some $B \in \mathcal{C}$, and then $F[B] \in ChMod_{\sim,\doteq}(SP)$. By stability, $F[A] \equiv_{\Sigma} F[B]$. Then, by Theorem 5.11, $F[A] \in ChMod_{\sim,\doteq}(SP)$. □

Corollary 5.14. *If $F \in WMod_{\sim,\doteq,\mathcal{C}}(SP' \to SP)$, $\equiv_{\Sigma}$ is* right factorizable *by $\sim_{\Sigma}$, the axioms of SP have equations in positive positions only and F is stable with respect to $\equiv_{\Sigma'}$ and $\equiv_{\Sigma}$, then $F \in WMod_{\sim,\doteq,Cl_{\equiv}(\mathcal{C})}(SP' \to SP)$.*

Theorem 5.13 and Corollary 5.14 are useful "amplification" results. They allow information gained from testing particular cases (here, the set of stubs $\mathcal{C}$) to be extrapolated to give information about cases that have not actually been tested. Theorem 5.13 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 5.13 are met. In Corollary 5.14, 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.

Example 5.15 (Extrapolating Test Results). Let $InvoiceFun$: Order×Stock → Invoice be a generic unit. Let $\mathcal{CO}$, $\mathcal{CS}$ be sets of models of Order and Stock respectively. Let $SFun \in \mathcal{CS}$ represent stocks as functions from product to quantity, and suppose $InvoiceFun$ is successfully tested as follows:

$$\forall O \in \mathcal{CO}, S \in \mathcal{CS} \cdot \bigwedge_{(\psi,T)\in\Psi_I} \mathit{InvoiceFun}[O,S] + \mathtt{IH} \models^{T}_{\sim_{\Sigma I},\hat{=}_{\Sigma I}} \psi$$

where IH, Ψ_I and ΣI are as in Example 4.5. According to Theorem 5.13 and Corollary 5.14 this corresponds to testing *InvoiceFun* using the whole class of models of ORDER and STOCK that are equivalent to models in $\mathcal{CO}$ and $\mathcal{CS}$ respectively. Now, let S' be a model of STOCK that is developed and tested separately. If we want to check whether $\mathit{InvoiceFun}[O,S']$ is a model of INVOICE with $O \in \mathcal{CO}$ but $S' \notin \mathcal{CS}$, we need to check whether S' is equivalent to a model in $\mathcal{CS}$. Suppose S' represents stocks as lists of pairs of product and quantity; clearly, S' may be equivalent to SFun. If it is, we can conclude that $\mathit{InvoiceFun}[O,S']$ is a checkable model of INVOICE. □

6 Testing from architectural specifications

This section is about testing modular systems, taking architectural specifications into account as structuring mechanisms. We also consider the use of *off-the-shelf* units. The framework provided by CASL architectural specifications ensures that units are independent: the use of a unit, for example in building another unit, can only take advantage of properties that appear in its specification. Therefore, styles of testing presented in Sections 4 and 5 can be applied to check individual units. Ways of *integrating* test results of individual units when they are used to form larger systems are addressed in this section.

Example 6.1 (Integrating Test Results). Consider the architectural specification:

arch spec INVOICESYSTEM =
units
NatAlg	:	NAT;
OrderFun	:	NAT → ORDER;
OrderAlg	=	*OrderFun*[*NatAlg*];
StockFun	:	NAT → STOCK;
StockAlg	=	*StockFun*[*NatAlg*];
InvoiceFun	:	ORDER × STOCK → INVOICE **given** *NatAlg*;

result *InvoiceFun*[*OrderAlg*,*StockAlg*]

(The "**given**" clause requires that both of the arguments of *InvoiceFun* are built using the *same* model of NAT, namely *NatAlg*.) Development of the modular system described by this specification can proceed top-down or bottom-up, and may reuse units that have already been developed and verified. Example 5.15 illustrates how the results of Section 5 can be used to extrapolate test results to avoid retesting. In a similar vein, suppose that development is strictly top-down and *InvoiceFun* is initially developed and tested with stubs *StubOrder* and *StubStock*. When *OrderAlg* and *StockAlg* are finally developed and successfully tested as models of ORDER and STOCK respectively, then if we can establish that *OrderAlg* ≡ *StubOrder* and *StockAlg* ≡ *StubStock*, we can conclude from Theorem 5.13 without needing to perform any further tests that *Invoice-Fun*[*OrderAlg*,*StockAlg*] is a model of INVOICE (provided we are willing to accept the stability hypothesis). Now, suppose that development is strictly bottom-up, and *OrderAlg* and *StockAlg* are individually developed and successfully tested as models of ORDER and STOCK respectively. One might think that it is appropriate to use these units to test an implementation of *InvoiceFun*, since testing of the combined system *InvoiceFun*[*OrderAlg*,*StockAlg*] will be performed later anyway. However, if *OrderAlg* and *StockAlg* have errors that have not been revealed by testing, then there is a chance that *InvoiceFun* will be rejected due to these undetected errors even if it is correct. It is therefore better to develop and test *InvoiceFun*, *OrderAlg* and *StockAlg* independently and then check again for errors when they are combined. □

In this section, we consider architectural specifications of the form:

$$\begin{array}{ll}\mathsf{units} & A : SP'';\\ & F : SP' \to SP\\ \mathsf{result} & F[A]\end{array}$$

where A is a unit that is a realization of SP'', F is a generic unit that is a realization of $SP' \to SP$, and $F[A]$ is the resulting unit term. Restricting attention to this simple special case allows us to focus on the main issue in integrating independently-developed modules, namely the interaction between a generic unit and its parameter. Our conclusions will apply to each of the occurrences of application of a generic unit to a parameter in a more complicated architectural specification such as the one in Example 6.1.

Notice that A is supposed to be a realisation of SP'' rather than the possibly different SP' which is the specification of F's parameter. This situation would arise if A is a so-called *off-the-shelf unit*. Such units may have more functionality than needed for their context of use. In particular, they may satisfy more properties than are actually required.

We have checked A against SP'', establishing $A \in ChMod_{\sim,\hat{=}}(SP'')$. Since we will be using A in a context where it is required to satisfy SP', we are interested in the question of whether $A \in ChMod_{\sim,\hat{=}}(SP')$. Therefore, we now consider conditions under which a checkable model A of SP'' is also a checkable model of SP'. The signature of SP'' is required to coincide with that of SP'; this guarantees that $F[A]$ is "statically" well-formed. However, the structure of the specification SP'' may not coincide with the structure of SP'. Also, the test sets in SP' and SP'' may differ even if the specifications otherwise coincide. The possible structural mismatch suggests that comparing the normal forms of the two specifications might help. Let $\boldsymbol{nf}(SP') = \langle \Sigma', \Psi' \rangle|_{Sig(SP')}$ and $\boldsymbol{nf}(SP'') = \langle \Sigma'', \Psi'' \rangle|_{Sig(SP'')}$, where $\Psi' \subseteq \{(\psi, T) \mid \psi \in Sen(\Sigma'),\ T \subseteq T_{\Sigma'}\}$ and $\Psi'' \subseteq \{(\psi, T) \mid \psi \in Sen(\Sigma''),\ T \subseteq T_{\Sigma''}\}$. Then we get the following result.

Theorem 6.2. *If $\sim$ and $\hat{=}$ are compatible, the axioms of SP' have only positive occurrences of $\forall$ and negative occurrences of $\exists$, and $\forall(\psi, T') \in \Psi' \cdot \exists(\psi, T'') \in \Psi'' \cdot T' \subseteq T''$, then $A \in ChMod_{\sim,\hat{=}}(SP'')$ implies $A \in ChMod_{\sim,\hat{=}}(SP')$.*

Proof. As $\sim$ and $\doteq$ are compatible families of equalities, $ChMod_{\sim,\doteq}(SP') = ChMod_{\sim,\doteq}(\boldsymbol{nf}(SP'))$ by Theorem 4.9. The same applies to SP''. Thus, we need to show that $A \in ChMod_{\sim,\doteq}(\boldsymbol{nf}(SP''))$ implies $A \in ChMod_{\sim,\doteq}(\boldsymbol{nf}(SP'))$. This follows from the fact that test sets in SP'' are bigger than test sets in SP' [Mac00b], using the assumptions on quantifiers, and that any axiom of SP' is also an axiom of SP''. $\square$

Theorem 6.2 relates to results in [Mac00a,Mac00c], regarding testing axioms with different test sets (Theorem 3.20 of [Mac00c]) and structured testing compared to testing from normal form (Corollary 5.36 there).

When the tester of F is different from the tester of A, then different approximate equalities can be chosen.

Theorem 6.3. *Let $\sim, \sim', \doteq, \doteq'$ be compatible families of equalities. If $\Psi' = \Psi''$, $\sim' \subseteq \sim$ and $\doteq' \supseteq \doteq$, then $A \in ChMod_{\sim',\doteq'}(SP'')$ implies $A \in ChMod_{\sim,\doteq}(SP')$.*

Proof. Again, this reduces to showing that $A \in ChMod_{\sim',\doteq'}(\boldsymbol{nf}(SP''))$ implies $A \in ChMod_{\sim,\doteq}(\boldsymbol{nf}(SP'))$. This follows from the fact that the pair of families of equalities $\sim', \doteq'$ is more accurate than the pair $\sim, \doteq$. $\square$

Theorem 6.3 relates to Theorem 3.27 in [Mac00c], although the implication is considered there for individual formulas only. The theorem illustrates the case in which A is tested against SP'' using more accurate equalities than the ones used to test F. If $\sim$ and $\sim'$ are complete, then the finer $\sim'$ is, the more accurate it is. Also, if $\doteq$ and $\doteq'$ are sound, then the coarser $\doteq'$ is, the more accurate it is. The following is a corollary of Theorems 6.2 and 6.3.

Corollary 6.4. *Let $\sim, \sim', \doteq, \doteq'$ be compatible families of equalities such that $\sim' \subseteq \sim$ and $\doteq' \supseteq \doteq$. If the axioms of SP' have only positive occurrences of $\forall$ and negative occurrences of $\exists$, and $\forall(\psi, T') \in \Psi' \cdot \exists(\psi, T'') \in \Psi'' \cdot T' \subseteq T''$, then $A \in ChMod_{\sim',\doteq'}(SP'')$ implies $A \in ChMod_{\sim,\doteq}(SP')$.* $\square$

One might expect a supplier of off-the-shelf units to use higher standards for testing than a user of such units. Then, larger test sets and more accurate equalities are indeed realistic expectations.

Example 6.5 (Off-the-shelf units). Consider a version of example 6.1 where $StockFun$: NAT $\rightarrow$ STOCK is replaced by *StockFun′* : NAT $\rightarrow$ STOCK′, where the signatures of STOCK and STOCK′ coincide but the test sets of STOCK are a subset of the test sets of STOCK′. Then, by Theorem 6.2, if *StockAlg* is a checkable model of STOCK′, it is certainly a checkable model of STOCK, making it an appropriate argument for *InvoiceFun*. Now, suppose instead that the equality `==` on *Stock* defined in Example 3.5 is used to test both *StockAlg* and *InvoiceFun*, but the list of products required as argument by `==` differs. Then the axioms of *StockAlg* and *InvoiceFun* are tested with different equalities on *Stock*. However, if the assumptions on Theorem 6.3 are met, then again if *StockAlg* is a checkable model of STOCK′, it is an appropriate argument for *InvoiceFun*. $\square$

7 Concluding remarks

We have presented ideas relating to testing modular systems against CASL-style architectural specifications. We focus on architectural specifications containing unit terms formed by instantiating generic units. 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 Section 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 Section 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. Finally, (3) is addressed in Section 6, where the aim is to take advantage of tests already performed and allow for independent development. Previous results are generalised in order to deal with off-the-shelf units, which may have more functionality than is required.

As further work, we plan to look into circumstances under which the stability assumption 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 and to unit terms built using operations other than instantiation of generic units. Finally, a general method of applying the ideas presented along with practical case studies are needed.

References

[ABK$^+$] 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.

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

[BBL97] G. Bernot, L. Bouaziz, and P. LeGall. A theory of probabilistic functional testing. *Proc. Intl. Conf. on Software Engineering*, Boston (1997).

[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).

[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).

[BGM91] G. Bernot, M.-C. Gaudel and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6:387–405 (1991).

[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 implementation. *Science of Computer and Programming*, 25:149–186 (1995).

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

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. *Proc. FME'93.* Springer LNCS 670 (1993).

[Don97] M.R. Donat. Automating formal specification-based testing. *Proc. TAPSOFT'97*, Lille. Springer LNCS 1214 (1997).

[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).

[GJ98] M-C. Gaudel and P.R. James. Testing abstract data types and processes: A unifying theory. *Formal Aspects of Computing*, 10:436–451 (1998).

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

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

[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).

[Mar91] B. Marre. Toward automatic test data selection using algebraic specifications and logic programming. *Proc. 8th Intl. Conf. on Logic Programming*, Paris. MIT Press (1991).

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

[SST92] D. Sannella, S. Sokołowski and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Informatica* 29:689–736 (1992).

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

Appendix: Normal form of a structured specification

Dear referee: This appendix will not be included in the published version of the paper.

The following is the usual procedure for normalizing structured specifications – see [BCH99] – modified to deal with structured specifications with testing interface. The aim is to tranform a structured specification with testing interface SP into a specification $\boldsymbol{nf}(SP)$ of the form $\langle \Sigma', \Psi' \rangle|_{\Sigma}$ having the same signature and the same class of models

The *symbols* of a specification includes both visible and hidden symbols appropriately renamed to avoid name clashes.

Definition A.1 (Symbols). *The* symbols *of a structured specification SP are defined as follows.*

1. *If $SP = \langle \Sigma, \Psi \rangle$, then $Sym(SP) = \Sigma$*
2. *If $SP = SP_1 \cup SP_2$, then $Sym(SP) = Sym(SP_1) +_{Sig(SP)} Sym(SP_2)$ (see Figure 1)*
3. *If $SP = \textsf{translate } SP' \textsf{ by } \sigma$, then $Sym(SP) = PO(Sig(SP') \hookrightarrow Sym(SP'), \sigma)$, where $\sigma : Sig(SP') \to Sig(SP)$ (see Figure 1)*
4. *If $SP = SP'|_{\Sigma}$, then $Sym(SP) = Sym(SP')$*

where the diagrams in Figure 1 are pushout constructions chosen to rename all hidden symbols so that $Sig(SP) \subseteq Sym(SP)$.

$$\begin{tikzcd}
Sig(SP) \arrow[r, hook] \arrow[d, hook] & Sym(SP_2) \arrow[d, dashed, "in_2"] \\
Sym(SP_1) \arrow[r, dashed, "in_1"] & Sym(SP_1)+_{Sig(SP)} Sym(SP_2)
\end{tikzcd}
\qquad
\begin{tikzcd}
Sig(SP') \arrow[r, "\sigma"] \arrow[d, hook] & Sig(SP) \arrow[d, hook, dashed] \\
Sym(SP') \arrow[r, dashed, "\varsigma"] & PO(Sig(SP') \hookrightarrow Sym(SP'), \sigma)
\end{tikzcd}$$

Fig. 1. $Sym(SP_1 \cup SP_2)$ and $Sym(\textsf{translate } SP' \textsf{ by } \sigma)$ respectively.

The normal form of a specification SP is a basic specification restricted by the export operator.

Definition A.2 (Normal Form). *Let SP be a structured specification. Its normal form $\mathbf{nf}(SP)$ is a specification of the form $\langle Sym(SP), \Psi \rangle|_{Sig(SP)}$, where $\Psi \subseteq \{(\psi, T) \mid \psi \in Sen(Sym(SP))$ and $T \subseteq T_{Sym(SP)}\}$, defined as follows.*

1. *If $SP = \langle \Sigma, \Psi \rangle$, then $\mathbf{nf}(SP) = \langle \Sigma, \Psi \rangle|_{\Sigma}$*
2. *If $SP = SP_1 \cup SP_2$ and $\mathbf{nf}(SP_i) = \langle Sym(SP_i), \Psi_i \rangle|_{Sig(SP_i)}$ $i = 1, 2$, then $\mathbf{nf}(SP) = \langle Sym(SP), in_1(\Psi_1) \cup in_2(\Psi_2) \rangle|_{Sig(SP)}$ (see Figure 1)*
3. *If $SP = \textsf{translate } SP' \textsf{ by } \sigma$ and $\mathbf{nf}(SP') = \langle Sym(SP'), \Psi' \rangle|_{Sig(SP')}$, then $\mathbf{nf}(SP) = \langle Sym(SP), \varsigma(\Psi') \rangle|_{Sig(SP)}$ (see Figure 1)*
4. *If $SP = SP'|_{\Sigma}$ and $\mathbf{nf}(SP') = \langle Sym(SP'), \Psi' \rangle|_{Sig(SP')}$, then $\mathbf{nf}(SP) = \langle Sym(SP'), \Psi' \rangle|_{Sig(SP)}$*

where ς, in_1, in_2 are extended to translate pairs of formulas and sets of terms.