Correctness of generic modules

Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Reif, Wolfgang. 1992. "Correctness of generic modules." In Logical Foundations of Computer Science — Tver '92: Second International Symposium, Tver, Russia, July 20–24, 1992; proceedings, edited by Anil Nerode and Mikhail Taitslin, 406–17. Berlin: Springer. https://doi.org/10.1007/BFb0023893.

Correctness of Generic Modules

Wolfgang Reif *
University of Karlsruhe
reif@ira.uka.de

Abstract

Modularization is the central design paradigm for large software systems, and the correctness of modules is one of the basic building blocks for the verification of large systems. In this paper we investigate the correctness problem for *generic* modules. The export- and import interfaces of a generic module are *parameterized* first-order specifications, and its implementation is written in an imperative programming language. We define the semantics and the correctness of generic modules, and present a logical characterization of correctness in terms of Dynamic Logic. This characterization paves the way for automated verification of generic modules in the KIV system (Karlsruhe Interactive Verifier), a tactical theorem prover for Dynamic Logic.

Key words: Program verification, module correctness, refinement of parameterized specifications, Dynamic Logic.

1 Introduction

The formal specification and systematic construction of verified software is one of the most prominent applications of logics of programs. In the literature, logics of programs have been used to formalize and to prove properties of programs such as partial correctness ([Ho 69]), termination, total correctness ([Bu 74]), program inclusion ([BK 84]), or more general relations between programs ([BM 79]). They have also been used as logical frameworks for program synthesis ([Gr 81], [CM 88], [HRS 91]). From a software engineering point of view, the majority of these applications refer to a logical analysis of *programming in the small*. So far, however, only a small number of approaches deal with *programming in*

---

* Author's address: Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, Postfach 6980, W-7500 Karlsruhe, FRG, Tel. +721-608-4245. This research was partly sponsored by the BMFT-project KORSO.

the large. In this field, modularization is the central methodological paradigm, and the correctness of software modules is one of the basic building blocks for the verification of large systems.

In this paper we address this problem, and present a new approach to module correctness. We consider *generic* or *parameterized modules*, with formally specified export- and import interfaces and a procedural implementation of the export specification in terms of the import specification. The specification language is full first-order logic. We give a characterization of module correctness in terms of Dynamic Logic ([Ha 79], [Go 82]). This characterization provides a more general solution to the correctness problem than approaches based on Hoare's logic ([Ho 72], [AS 89]), or "classical" approaches based on universal algebra and category theory ([GM 82], [EM 85], [BEP 87], [EM 90]). Additionally, the characterization of module correctness using DL paves the way for automated module verification. A method based on this result has been implemented and successfully tested in the KIV system (Karlsruhe Interactive Verifier, [HRS 88], [HRS 89], [HRS 90]) which is a tactical theorem prover for DL and can be used to verify and develop imperative programs.

The paper is organized as follows: In the next section we define specifications, generic modules, and give a formal account of semantics and correctness. The concepts are illustrated by an example, which will be used throughout the paper. In section 3 we give a brief overview of DL. Section 4 presents the characterization of module correctness in DL, and in section 5 we draw some conclusions.

2 Specifications and Generic Modules

2.1 Parameterized Specifications

The syntax of parameterized specifications is illustrated by the following description of finite sets over an arbitrary partially ordered domain. The operations are the empty set $\emptyset$, the insertion of a data element (insert), the membership predicate ($\in$), and an ordering ($<<$) on sets, being a variant of the multiset ordering. The ordering on the data domain is $<$:

parameterized specification Sets_over_Partial_order

parameters	Partial_order	(* see below)
sorts	set	
functions:	$\emptyset : \rightarrow$ set	
	insert : data $\times$ set $\rightarrow$ set	
predicates	infix $\in$: data $\times$ set	
	infix $<<$: set $\times$ set	
vars	n, m : set	
axioms	$\neg\ x \in \emptyset$	
	$x \in \text{insert}(y, m) \leftrightarrow (x = y \vee x \in m)$	
	$n = m \leftrightarrow \forall x.\ (x \in n \leftrightarrow x \in m)$	

$n << m \leftrightarrow \forall x.\,(x \in n \rightarrow \exists y.\,(y \in m \wedge x < y))$

end

parameter specification Partial_order
sorts data
predicates infix $<$: data × data
vars x, y, z : data
axioms $\neg\, x < x$
$x < y \wedge y < z \rightarrow x < z$

end

Parameterized specifications are based on signatures with sorts, and operation symbols (functions and predicates), on variable declarations and sets of first-order formulas with tacit universal closure (axioms). In the example Partial_order is the *parameter* specification, Sets_over_Partial_order the *target* specification. Correspondingly, we distinguish between parameter and target sorts, and parameter and target operations. Formally, we write SIG = $(S \cup S_p, OP \cup OP_p)$ for a signature with parameter sorts S_p, target sorts S, parameter operations OP_p, and target operations OP. In contrast to other approaches in the literature ([GTW 78], [EM 85], [Pa 90]) the axioms are not restricted to universal Horn clauses or equations. This increases the flexibility of the specification language for practical applications. Concerning the semantics of a parameterized specification SPEC over a signature SIG, the only information we need in this paper, is that the semantics is based on the class of its *parameter generated* models PGen(SPEC). This is the class of all SIG-algebras, Alg(SIG), that satisfy the axioms of SPEC, and meet the following constraint: Each element of any target carrier can be (finitely) constructed from the parameter carriers using the target functions. For a discussion of consistency, monomorphism, correctness and related topics we refer to [Re 91].

2.2 Generic Modules

A *module* is a triple consisting of an *export specification*, an *implementation* and a *mapping*. The export specification provides sorts and operations, that are visible to the environment, and that may be used there as elementary operations. Its axioms describe the external behaviour of the module. The implementation is a collection of procedures *and* import specifications. The import specifications describe the data the procedures operate upon. The mapping sets up the correspondence between the sorts and operations of the export specification, and the sorts and procedures of the implementation. It indicates which sort or operation of the export specification is implemented by which sort or procedure of the implementation. A module is called *generic* if the export- and import specifications are parameterized specifications sharing the same parameter specification. The parameter specification is the generic part of the module. We illustrate these concepts with a module implementing *sets* by *lists*: Sets_over_Partial_order is implemented by list processing procedures given in ListProc, which in turn are based on a specification of lists:

generic module Set_ListProc

exporting	Sets_over_Partial_order			
implementation	ListProc	(*see below)		
mapping	list	implements	set	
	EMPTY	implements	∅	
	INSERT	implements	insert	
	MEMBER	implements	∈	
	LESS	implements	<<	
	IDF	implements	=	for set

end

implementation ListProc

importspec Lists_over_Partial_order

declarations EMPTY ⇐ ..., INSERT ⇐ ..., MEMBER ⇐ ..., LESS ⇐ ..., IDF ⇐ ...

end

The mapping in Set_ListProc indicates which sort or operation is implemented by which sort or procedure of ListProc. The implementation ListProc imports a specification Lists_-over_Partial_order, with target sort *list* and parameter sort *data*, and provides declarations for the procedures that are referenced in the mapping of the module (EMPTY ⇐ ... etc.). Here we do not go into the details of the procedure declarations and the specification Lists_-over_Partial_order (they are given partly in section 4). The equality of the target sort *set* is implemented by a procedure IDF. This is required since a set may be represented in different ways. Multiple list representations denoting the same set are identified by IDF. Generally, the parameter sorts and operations are not mentioned in the mapping. They are treated in a standard way: each parameter sort and operation is mapped onto itself. This is possible since the export- and import specifications share the same parameter specification.

Roughly, the semantics of a generic module M is a partial function Sem(M), induced by the implementation of the module. It takes a parameter generated model of the import specifications (import algebra) and assigns to it a parameter generated model of the export signature (export algebra). The operations of the export algebra are basically the procedures of the implementation. A module is correct if Sem(M) is a total function, and if the export algebras not only are models of the export signature, but additionally satisfy the export specification. As a consequence, a correct module M has the following nice property: In order to provide a correct and executable implementation of M's export specification, it suffices to compose its implementation with any correct and executable implementation of its import specifications. Informally, a module is correct if the procedures of its implementation terminate and exhibit the behaviour specified for the export operations.

We are going to give a formal account of these notions. To simplify the presentation we will refer to a simple generic module *M* with export specification *Exp*, an implementation

Prog with a single import specification *Imp*, and a mapping *Map*. We assume that both Exp and Imp only have a single target sort s_e and s_i, respectively, and a single parameter sort s_p. Let $SIG_e = (\{s_e\} \cup \{s_p\}, OP_e \cup OP_p)$ the signature of Exp, $SIG_i = (\{s_i\} \cup \{s_p\}, OP_i \cup OP_p)$ the signature of Imp, and $SIG_p = (\{s_p\}, OP_p)$ the subsignature for the parameter. Let IDF implement the equality on s_e.

Definition (Semantics and correctness of a generic module)

The semantics of M, Sem(M), is a partial function which maps algebras from PGen(Imp) to PGen(SIG_e). We define its domain, and how the export algebra is constructed from the import algebra, simultaneously. Let $\mathcal{A} \in$ PGen(Imp). For every function symbol $f \in OP_e$ from SIG_e the semantics $[\![Map(f)]\!]$ of the corresponding procedure Map(f) in Prog is (by syntactic restriction of the programs) a partial function over $\mathcal{A}$. Let F = {Map(f): f a function symbol from OP_e} be the set of procedures associated with the function symbols. For a predicate symbol $f \in OP_e$ the semantics of Map(f) is a partial truth function. In this case let $[\![Map(f)]\!]$ denote the relation which holds for a given argument iff Map(f) terminates for it, and returns true. The functions and relations $[\![Map(f)]\!]$ serve as first tentative candidates for the interpretations of f. Next, with the help of the procedures in F, we define a pair of sets $(G_{\mathcal{A}}, A_{sp})$, which serve as the basis for the carrier sets of the intended export algebra. A_{sp} is the parameter carrier of $\mathcal{A}$, and $G_{\mathcal{A}}$ is defined as follows: Let $G_{\mathcal{A}}$ the set of all elements of A_{si} (the target carrier of $\mathcal{A}$) that can be constructed from A_{sp} by iterated application and combination of the procedures from F. If all the procedures from F and those for the predicate symbols terminate on $(G_{\mathcal{A}}, A_{sp})$ (which is not guaranteed per construction), we may define an intermediate parameter generated SIG_e-algebra $\mathcal{G}_{\mathcal{A}} := (G_{\mathcal{A}}, A_{sp}, [\![Map(f)]\!]_{f \in OP_e}, (f_{\mathcal{A}})_{f \in OP_p})$, where $f_{\mathcal{A}}$ is the interpretation of f in $\mathcal{A}$. Finally, we have to take into account, that different elements of $G_{\mathcal{A}}$, which are identified by IDF, are supposed to denote the same object in the export algebra. The semantics of the identification program IDF induces a binary relation $\equiv_{IDF}$ on $G_{\mathcal{A}}$ which holds for two elements if they are identified by IDF. Now we can define domain and value of Sem(M): Sem(M)($\mathcal{A}$) is defined if all the procedures of Prog and IDF terminate on $(G_{\mathcal{A}}, A_{sp})$, and if $\equiv_{IDF}$ is a congruence on $\mathcal{G}_{\mathcal{A}}$. Then Sem(M)($\mathcal{A}$) is the quotient of $\mathcal{G}_{\mathcal{A}}$ by $\equiv_{IDF}$.

M is *correct* iff Sem(M)($\mathcal{A}$) is defined for all $\mathcal{A} \in$ PGen(Imp) and Sem(M)($\mathcal{A}$) $\models$ Exp ■

Our aim is to translate the correctness of a module into proof obligations of Dynamic Logic. Before we do so, we give a brief overview of DL.

3 Dynamic Logic

Dynamic Logic ([Ha 79], [Go 82]) extends predicate logic (PL) by formulas $[\pi]\varphi$ ("box π φ") and $\langle\pi\rangle\varphi$ ("diamond π φ"), where π is a program, and φ is again a DL-formula. The

intuitive meaning of $[\pi]\varphi$ is: "if π terminates, φ holds after execution of π ". The formula $\langle\pi\rangle\varphi$ has to be read as: "π terminates and φ holds after execution of π ". The programs that may occur in such program formulas are taken from the following Pascal-like programming language. We consider the commands skip, abort (the never halting program), assignments, conditionals, while loops, local variables and mutually recursive procedures. With the command **var** $x_1=t_1, ..., x_n=t_n$ **in** α, the local variables x_i are introduced and initialized by t_i. The command α is the scope of the declarations. Procedure declarations $p_1 \Leftarrow \lambda(\mathbf{x_1} : \mathbf{y_1}).\alpha_1$, .., $p_n \Leftarrow \lambda(\mathbf{x_n} : \mathbf{y_n}).\alpha_n$, introduce the mutually recursive procedures p_i. The vectors $\mathbf{x}_i$ and $\mathbf{y}_i$ are the formal value- and reference parameters, respectively, and α_i is the procedure body. In the procedure call $p(\mathbf{t} : \mathbf{z})$, **t** and **z** are the actual parameters. A program π is a pair (decl | α) consisting of a command α and a procedure declaration decl, where the definitions for the procedure calls of α are looked up. If the declaration is clear from the context we will omit it and write $\langle\alpha\rangle\varphi$, instead of $\langle\pi\rangle\varphi$.

In DL many interesting properties of programs can be expressed. For PL-formulas φ and ψ, the formula $\varphi \rightarrow [\pi]\psi$ corresponds to the well known partial correctness assertion $\varphi\{\pi\}\psi$. Total correctness assertions, $\varphi \rightarrow \langle\pi\rangle\psi$, are obtained by using diamond formulas instead of box formulas. Other properties are non-termination $\neg\langle\pi\rangle$true, or the equivalence of two programs π and π' with respect to a program variable x: $\langle\pi\rangle x=x' \leftrightarrow \langle\pi'\rangle x=x'$. In order to express the correctness of a module we need a nondeterministic version of DL with random assignments x:=? and randomly initialized local variables **var** x = ? **in** α as additional commands. They randomly assign a value to x. They are not allowed in the implementation of a module, we only use them for descriptive purposes. Random assignments can be axiomatized by existential quantifiers: $\langle x := ?\rangle\varphi \leftrightarrow \exists x.\varphi$. Furthermore, in this indeterministic version of DL, parameter generatedness of a SIG-algebra can be expressed by a set of formulas, DL-IND(SIG), the set of DL instances of the structural induction principle over SIG (see [Re 91]).

4 A Characterization of Module Correctness with Dynamic Logic

Now we present a translation of the correctness of a generic module into a set of proof obligations formulated in DL. These proof obligations are necessary and sufficient for the correctness of the module. This result provides a general solution to the correctness problem for implementations of parameterized first-order specifications. Although our programs are written in an imperative language the result applies to a wider range of languages. We illustrate the translation with the help of the above Set_ListProc example and add comments dealing with the general case.

First we give two example definitions of the procedures in ListProc (implementation of Set_ListProc) which have been omitted in 2.2.

```
EMPTY ⇐ λ(: l1). l1 := nil                          (* outputs the empty list *)
INSERT ⇐ λ(x, l1 : l2).                             (* inserts x in l1 if not yet there*)
        var   l3 = l1  in
        if l1 = nil then l2 := cons(x, nil)
        else  if x = car(l1) then  l2 := l1
              else INSERT(x, cdr(l1): l3) ; l2 := cons(car(l1), l3) fi fi
```

The procedures EMPTY and INSERT compute on List_over_Partial_order, which provides the usual list operations *nil* (empty list), *cons* (inserting an element at the front), the selectors *car* and *cdr* (first element and rest of a list, rsp.), and list variables l_1, l_2, l_3. Due to space limitations we omit the specification.

For the general case, we refer in the sequel to a module M, for which we adopt the same assumptions and the notation used in 2.2: M is a simple generic module with export specification Exp, the implementation Prog with a single import specification Imp, and a mapping Map. Both Exp and Imp are supposed to have a single target sort s_e and s_i, respectively, and a single parameter sort s_p. $SIG_e = (\{s_e\} \cup \{s_p\}, OP_e \cup OP_p)$ is the signature of Exp, $SIG_i = (\{s_i\} \cup \{s_p\}, OP_i \cup OP_p)$ the signature of Imp, and $SIG_p = (\{s_p\}, OP_p)$ the subsignature for the parameter. IDF implements the equality on s_e. Our sample module Set_ListProc is of this form, and whenever concrete examples are given, M refers to Set_ListProc.

The Translation

In a first step we define for the target sort s_e of the export signature a *restriction* R. The formula R is intended to describe the set of data (of sort $s_i = Map(s_e)$) that can be constructed from the parameter elements (of sort s_p) by applying the procedures of the implementation (see $G_{\mathcal{A}}$ in 2.2). Relative to a restriction R and the module M, four sets of proof obligations (verification conditions in DL, with tacit universal closure) are defined: $VC_T(R, M)$, $VC_C(R, M)$, $VC_{EXP}(R, M)$ and $VC_G(R, M)$. These sets will guarantee M's correctness.

Choosing a Restriction R

In our example Set_ListProc the set of lists that can be constructed by the procedures EMPTY and INSERT (the implementations of the constructors), are the lists without repetitions (no duplicates). Therefore, $R(l_1)$ is chosen to be true for a list l_1 iff l_1 has no duplicates. In the following concrete proof obligations, R refers to this formula.

Proof Obligations $VC_T(R, M)$

Generally, the operations of the export specification have to be interpreted by total functions. However, the procedures of the implementation of M may fail to terminate. Therefore, we require that these procedures terminate at least for inputs taken from R and/or the parameter. This is done in $VC_T(R, M)$. In our example this means, that EMPTY and INSERT terminate for restricted inputs, and produce outputs belonging to R. Furthermore,

the predicates MEMBER and LESS terminate for restricted inputs. This is expressed by the following conditions:

$VC_T(R, M) = \{$

$\langle EMPTY(: l_1)\rangle\, R(l_1)$,

$R(l_1) \rightarrow \langle INSERT(x, l_1 : l_2)\rangle\, R(l_2)$,

$R(l_1) \rightarrow \langle MEMBER(x, l_1 : b)\rangle$ true, (b is a boolean variable)

$R(l_1) \wedge R(l_2) \rightarrow \langle LESS(l_1, l_2 : b)\rangle$ true$\}$

Proof Obligations $VC_C(R, M)$

This set of proof obligations guarantees that the identification procedure IDF of M, which implements equality of the target sort s_e, terminates and behaves like a congruence on R with respect to the implementations of the function- and predicate symbols. In the Set_ListProc example, IDF is a program which identifies two lists l_1 and l_2 (by returning true) iff l_1 and l_2 only differ in the order of their elements. As examples we present the conditions for termination and reflexivity, and omit the conditions for symmetry, transitivity and compatibility:

$R(l_1) \wedge R(l_2) \rightarrow \langle IDF(l_1, l_2 : b)\rangle$ true (termination)

$R(l_1) \rightarrow \langle IDF(l_1, l_1 : b)\rangle\, b = true$ (reflexivity)

Proof Obligations $VC_{EXP}(R, M)$

The conditions $VC_T(R, M)$ and $VC_C(R, M)$ imply that the semantics Sem(M) is defined for all models of PGen(Imp). The proof obligations $VC_{EXP}(R, M)$ guarantee that the procedures of the implementation exhibit the behaviour specified in the export specification. $VC_{EXP}(R, M)$ results from translating the export specification (exclusive of the parameter specification) to DL-formulas over the import signature: parameter variables, -operations and -equality are mapped onto themselves. Target variables of the export are translated to target variables of the import. A target operation f is translated to a call of the corresponding procedure Map(f). The equality over the target sort is translated to a call of IDF. The propositional connectives remain unchanged, and the quantifiers are treated as follows: For a parameter variable x, $\forall x.\varphi$ and $\exists x.\varphi$ are translated to $\forall x.\varphi'$ and $\exists x.\varphi'$, respectively, where φ' is the translation of φ. For a target variable m, $\forall m.\varphi$ and $\exists m.\varphi$ are translated to $\forall l_1.\ (R(l_1) \rightarrow \varphi')$ and $\exists l_1.\ (R(l_1) \wedge \varphi')$, respectively, where l_1 and φ' are the translations of m and φ. As examples we give the translations of two axioms from Set_ListProc (which both are implicitly, universally quantified):

$x \in insert(y, m) \leftrightarrow (x = y \vee x \in m)$ becomes:

$R(l_1) \rightarrow (\langle INSERT(y, l_1 : l_2)\,;\, MEMBER(x, l_2 : b)\rangle\, b = true \leftrightarrow$

$(x = y \vee \langle MEMBER(x, l_1 : b)\rangle\, b = true))$

$n = m \leftrightarrow \forall x.\,(x \in n \leftrightarrow x \in m)$ becomes:

$R(l_1) \wedge R(l_2) \rightarrow$

$(\langle IDF(l_1, l_2 : b)\rangle\, b = true \leftrightarrow$

$\forall x.\,(\langle MEMBER(x, l_1 : b)\rangle\, b = true \leftrightarrow \langle MEMBER(x, l_2 : b)\rangle\, b = true))$

Proof Obligations $VC_G(R, M)$

The last set of proof obligations guarantees that the restriction R indeed denotes the set of data that can be constructed by the procedures implementing the constructors of the export specification. The conditions $VC_T(R, M)$ already imply that R denotes a superset of the intended set. Therefore, it suffices to make sure that every data belonging to R can actually be constructed by the implementation. Unfortunately, in general this requirement is neither expressible in first-order logic nor in deterministic DL ([ST 83]). This is the reason why we added random assignments to the programming language. In this nondeterministic version of DL there exists (uniformly in M) a formula R_M which can be proved to describe exactly the set of input data constructible by the implementation. In the Set_ListProc example $R_M(l_1)$ is the DL-formula $\langle decl_M \mid r_M(l_1:)\rangle$true, where $decl_M$ contains the following declaration of the nondeterministic procedure r_M:

$r_M \Leftarrow \lambda(l_1:).$ **var** $l_2 = l_1$ **in**
EMPTY(: l_2) ; **if** $l_2 = l_1$ **then skip**
else **var** $l_2 = ?, x = ?, l_3 = l_1$ **in**
$r_M(l_2:)$; INSERT(x, $l_2 : l_3$) ; **if** $l_3 = l_1$ **then skip else abort fi fi**

The formula $R_M(l_1)$ holds iff r_M terminates for l_1. The procedure r_M terminates for the list l_1 iff l_1 is the result of EMPTY or if an x and a l_2 can be guessed such that r_M terminates on l_2, and l_1 can be constructed from x and l_2 by INSERT. Hence, we define $VC_G(R, M) = \{ R(l_1) \rightarrow R_M(l_1) \}$.

This completes the collection of proof obligations. We set $VC(R, M) = VC_T(R, M) \cup VC_C(R, M) \cup VC_{EXP}(R, M) \cup VC_G(R, M)$. Then the following theorem holds:

Theorem (Characterization of the correctness of generic modules)

Let M a generic module with import specification Imp over the signature SIG_i.

M correct $\Leftrightarrow$ there exists a restriction R such that: $Imp \cup DL\text{-}IND(SIG_i) \models VC(R, M)$.

Proof: see [Re 91] ■

The "if"-part of the theorem provides a practically applicable proof method to establish module correctness: Verifying a module M requires to invent a restriction R such that the proof obligations VC(R, M) follow from the import specification enriched by structural induction DL-IND(SIG_i). Note that the models of DL-IND(SIG_i) are exactly the parameter generated SIG_i-algebras (see 3). The "only-if"-part of the theorem states that every correct module M can be tackled this way. It guarantees the existence of a restriction R with the desired property. Actually, this restriction exists uniformly in M. We exploited this fact already to define $VC_G(R, M)$ where we used the uniform restriction R_M. If we replace R by R_M, the condition $VC_G(R_M, M)$ becomes a tautology. We obtain the following corollary:

Corollary (Correctness using R_M)

Let M a generic module with import specification Imp over the signature SIG_i.

M correct $\Leftrightarrow$

$Imp \cup DL\text{-}IND(SIG_i) \models VC_T(R_M, M) \cup VC_C(R_M, M) \cup VC_{EXP}(R_M, M)$.

Proof: see [Re 91] ■

Although, the corollary looks simpler than the theorem, and although it avoids the invention of a restriction R, the theorem is more useful for practical module verification. Experience shows that proofs involving R_M are much harder than those involving an intuitively chosen restriction. With this in mind the considerable advantage of the theorem is that it minimizes the proof obligations involving R_M.

Finally, we show that $VC_G(R, M)$ can be dropped for generic modules with a restricted form of export specifications. If SIG_e is the export signature and S_e the set of its target sorts, then we call $\forall_{S_e}L(SIG_e)$ the universal first-order formulas with respect to S_e. By this we mean the class of closed first-order formulas of the following shape: $Q_1...Q_n.\ \varphi$, where none of the quantifier prefixes Q_i is an existential one over a target sort, and φ must not contain any quantifications over target sorts. Quantifications over parameter sorts are allowed both in the prefixes and in φ. Now the following theorem holds.

Theorem (Correctness for universal generic modules)

Let M a generic module with export specification Exp over SIG_e and import specification Imp over SIG_i. Let $Exp \subseteq \forall_{S_e}L(SIG_e)$.

M correct $\Leftrightarrow$ there exists a restriction R such that:

$$Imp \cup DL\text{-}IND(SIG_i) \models VC_T(R, M) \cup VC_C(R, M) \cup VC_{EXP}(R, M).$$

Proof: see [Re 91] ■

The theorem shows that the proof obligations $VC_G(R, M)$ are related to existential quantifiers in the export specification, and may be dropped if there are none. The reason for that is based on the following observation: Let $\exists m.\varphi$ be a formula of the export specification of M with an existential quantification over a target sort. According to the above corollary (using the uniform restriction R_M) this leads among others to a proof obligation $\exists l_1.\ (R_M(l_1) \wedge \varphi')$ where φ' is the translation of φ according to the definition of VC_{EXP}. However, the "if"-part of the theorem under consideration refers to a restriction R which may differ from R_M, yielding the proof obligation $\exists l_1.\ (R(l_1) \wedge \varphi')$. Without the constraint $R(l_1) \rightarrow R_M(l_1)$ it is possible, that the latter obligation, and actually all the obligations in VC(R, M) - VC_G(R, M), are entailed by Imp $\cup$ DL-IND(SIG_i), whereas the former is not. This means that Imp $\cup$ DL-IND(SIG_i) entails $VC_T(R, M) \cup VC_C(R, M) \cup VC_{EXP}(R, M)$ but M is not correct (by the corollary). Therefore, $VC_G(R, M)$ is indeed required in context of existential quantifiers. However, the argument does not hold for universally quantified formulas $\forall m.\varphi$. If we argue as before, we get the two obligations $\forall l_1.(R_M(l_1) \rightarrow \varphi')$ and $\forall l_1.(R(l_1) \rightarrow \varphi')$. In this case, the obligations for termination $VC_T(R, M)$ imply the constraint $R_M(l_1) \rightarrow R_M(l_1)$. Therefore, the aforementioned conflicting situation is impossible: Imp $\cup$ DL-IND(SIG_i) entails the former obligation whenever it entails the latter.

In the Set_ListProc example, the export specification Sets_over_Partial_order is indeed universal with respect to the target sort *set*. Therefore, by the last theorem, we could have left out $VC_G(R, M)$ from our discussion of the proof obligations. We have included it, however, to illustrate the general case.

5 Conclusion

We have presented a simple concept for generic software modules. The interfaces of a generic module are parameterized full first-order specifications, and the implementation of the export interface in terms of the import interface is written in an imperative programming language. We have given a formal semantics of generic modules based on parameter generated algebras, and a formal notion of correctness. The main result is a characterization of module correctness in terms of Dynamic Logic. This characterization provides a general solution to the correctness problem of implementations of parameterized first-order specifications. This result has been turned into a proof method for automated module verification in the KIV system. It has been successfully tested on a number of case studies and is currently applied in an industrial context to produce verified software for a national radio station and verified access control software for nuclear power plants.

References

[AS 89] Antoniou, G.,Sperschneider, V., On The Verification of Modules, CSL 89, Springer LNCS 440, pp. 16-35

[BEP 87] Blum, E.K., Ehrig, H., Parisi-Presicce, F., Algebraic Specification of Modules and Their Basic Interconnections, Journal of Computer and System Sciences 34 (1987), pp. 293-339

[BM 79] Boyer, R., Moore, J, A Computational Logic, Academic Press, New York 1979

[BK 84] Bergstra, J., Klop, J., Proving Program Inclusion Using HOARE's Logic, Theoretical Computer Science 30 (1984), pp. 1-48

[Bu 74] R. Burstall, Program Proving as Hand Simulation with a little Induction, Information Processing 74, North-Holland Publishing Company (1974)

[CM 88] Chandy, K.M., Misra, J., Parallel Program Design, A Foundation. Addison Wesley 1988

[EM 85] Ehrig, H., Mahr, B., Fundamentals of Algebraic Specification 1, Equations and Initial Semantics, EATCS Monographs on Theoretical Computer Science, Vol. 6, Springer 1985

[EM 90] Ehrig, H., Mahr, B., Fundamentals of Algebraic Specification 2, Module Specifications and Constraints, EATCS Monographs on Theoretical Computer Science, Vol. 21, Springer 1990

[GM82] Goguen, J., Meseguer, J., Universal Realization, Persistent Interconnection and Implementation of Abstract Modules, Proc. 9th ICALP, 1982, Springer LNCS 45, 265-281

[GTW 78] Goguen, J., Thatcher, J., Wagner, E., An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, Current Trends in Programming Methodology IV, Yeh, R. (Ed.), Prentice-Hall, Englewood Cliffs, 1978, pp. 80-149

[Go 82] Goldblatt, R., Axiomatising the Logic of Computer Programming, Springer LNCS 130

[Gr 81] Gries, D. The Science of Programming, Springer-Verlag (1981)

[Ha 79] Harel, D., First-Order Dynamic Logic, Springer, LNCS 68, 1979

[Ho 69] Hoare, C.A.R., An Axiomatic Basis for Computer Programming, Comm. ACM 12 (1969)

[Ho 72] Hoare, C.A.R., Proof of Correctness of Data Representations, Acta Informatica 1,4 (1972), pp. 271-281

[HRS 88] Heisel, M., Reif, W., Stephan, W., Implementing Verification Strategies in the KIV-System, Proc. 9th International Conference on Automated Deduction, E. Lusk, R. Overbeek (eds), Springer LNCS 310 (1988), pp. 131-140

[HRS 89] Heisel, M., Reif, W., Stephan, W., A Dynamic Logic for Program Verification, Meyer, A., Taitslin, M., Logic at Botik 1989, Pereslavl-Zalessky, USSR, Springer LNCS 363, pp. 134-145

[HRS 90] Heisel, M., Reif, W., Stephan, W., Tactical Theorem Proving in Program Verification, 10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 1990, Springer LNCS 449, pp. 117-131

[HRS 91] Heisel, M., Reif, W., Stephan, W., Formal Software Development in the KIV System, in Automating Software Design, Lowry McCartney (eds), AAAI press 1991, and Proc. Workshop on Automating Software Design, IJCAI-89, Kestrel Institute, Palo Alto (1989), pp. 115 - 124.

[Pa 90] Padawitz, P., Horn Logic and Rewriting for Functional and Logic Program Design, Universität Passau, Fakultät für Mathematik und Informatik, MIP-9002, März 1990

[Re 91] Reif, W., Correctness of Specifications and Generic Modules, Dissertation, University of Karlsruhe, 1991 (in German)

[ST 83] Stolboushkin, A.P., Taitslin, M.A., Deterministic dynamic logic is strictly weaker than dynamic logic, Information and Control 57, pp. 48-55, 1983