Checking well-formedness of pure-method specifications

Arsenii Rudich[1], Ádám Darvas[1], and Peter Müller[2]

[1] ETH Zurich, Switzerland, {arsenii.rudich,adam.darvas}@inf.ethz.ch
[2] Microsoft Research, USA, mueller@microsoft.com

Abstract. Contract languages such as JML and Spec# specify invariants and pre- and postconditions using side-effect free expressions of the programming language, in particular, pure methods. For such contracts to be meaningful, they must be well-formed: First, they must respect the partiality of operations, for instance, the preconditions of pure methods used in the contract. Second, they must enable a consistent encoding of pure methods in a program logic, which requires that their specifications are satisfiable and that recursive specifications are well-founded.
This paper presents a technique to check well-formedness of contracts. We give proof obligations that are sufficient to guarantee the existence of a model for the specification of pure methods. We improve over earlier work by providing a systematic solution including a soundness result and by supporting more forms of recursive specifications. Our technique has been implemented in the Spec# programming system.

1 Introduction

Contract languages such as the Java Modeling Language (JML) [21] and Spec# [2] specify invariants and pre- and postconditions using side-effect free expressions of the programming language. While contract languages are natural for programmers, they pose various challenges when contracts are encoded in the logic of a program verifier or theorem prover, especially when contracts use pure (side-effect free) methods [13]. This paper addresses two challenges related to pure-method specifications.

The first challenge is how to ensure that a specification is *well-defined*, that is, that all partial operations are applied within their domain. For instance method calls are well-defined only for non-null receivers and when the precondition of the method is satisfied. This challenge can be solved by encoding partial functions as under-specified total functions [15]. However, it has been argued that such an encoding is counter-intuitive for programmers, is not well-suited for runtime assertion checking, and assigns meaning to bogus contracts instead of having them rejected by a verifier [8]. Another solution is the use of 3-valued logic, such as LPF [3]. However, 3-valued logic is typically not supported by the theorem provers that are used in program verifiers. We present a technique based on 2-valued logic to check whether a specification satisfies all partiality constraints. If the check fails, the specification is rejected.

```
interface Sequence {
  [Ghost] int Length;

  invariant Length >= 0;
  invariant IsEmpty() ==> Length == 0;
  invariant !IsEmpty() ==> Length == Rest().Length + 1;

  [Pure][Measure=Length] int Count(Object c)
     requires !IsEmpty();
     ensures result >= 0;
     ensures result == (GetFirst() == c ? 1 : 0) +
                       (Rest().IsEmpty() ? 0 : Rest().Count(c));
  [Pure] bool IsEmpty();
  [Pure] Object GetFirst()
     requires !IsEmpty();
  [Pure] Sequence Rest()
     requires !IsEmpty();
     ensures result != null;

  // other methods and specifications omitted
}
```

Fig. 1. Specification of interface `Sequence`. We use a notation similar to Spec#, which is an extension of C#. The **`Pure`** attribute marks a method to be side-effect free; pre- and postconditions are attached to methods by **`requires`** and **`ensures`** clauses, respectively. Invariants are specified in **`invariant`** clauses; in postconditions, **`result`** denotes the return value of methods. User-specified recursion measures are given by the **`Measure`** attribute. Fields marked with the **`Ghost`** attribute are specification-only.

The second challenge is how to ensure that a specification is consistent. In order to reason about contracts that contain pure-method calls, pure methods must be encoded in the logic of the program verifier. This is typically done by introducing an uninterpreted function symbol for each pure method m, whose properties are axiomatized based on m's contract and object invariants [10, 13]. A specification is *consistent* if this axiomatization is free from contradictions. Consistency is crucial for soundness. We present a technique to check consistency by showing that the contracts of pure methods are satisfiable and well-founded if they are recursive. If the consistency check fails, the specification is rejected.

An inconsistent specification of a method m is not necessarily detected during the verification of m's implementation [13]: (1) m might be abstract; (2) partial correctness logics allow one to verify m w.r.t. an unsatisfiable specification if m's implementation does not terminate; (3) any implementation could be trivially verified based on inconsistent axioms stemming from inconsistent pure-method specifications; this is especially true for recursion, when the axiom for m is needed to verify its implementation. These reasons justify the need for verifying consistency of specifications independently of implementations.

We illustrate these challenges by the interface `Sequence` in Fig. 1. It contains pure methods to query whether the sequence is empty, and to get the first element and the rest of the sequence. Method `Count` returns the number of occurrences of its parameter in the sequence. The interface contains the specification-only ghost field `Length`, which represents the length of the sequence. The interface is equipped with method specifications and invariants specifying `Length`.

We call a specification *well-formed* if it is well-defined and consistent. The main difficulty in the checking of well-formedness lies in the subtle dependencies between the specification elements. For instance, to be able to show that the expression `Rest().Count(c)` in `Count`'s postcondition is well-defined, the guarding condition `!Rest().IsEmpty()`, the precondition of `Count`, and the contract of `Rest` are needed. These specification elements together allow one to conclude that the receiver is not null and that the preconditions of `Rest` and `Count` are satisfied. That is, we need the specification of (axioms for) some pure methods to prove the well-definedness of other pure methods.

The second challenge is illustrated by the specification of method `Count`. Consistency requires that there actually is a result value for each call to `Count`. This would not be the case, for instance, if the first postcondition required `result` to be strictly positive. Since the specification of `Count` is recursive, proving the existence of a result value relies on the specification of `Count`. Using this specification is sound since the recursion in `Count`'s specification is well-founded: the first and third invariant, and the precondition of `Count` guarantee that the sequence is finite, and the guarding condition together with the precondition of `Count` and the third invariant guarantees that we recurse on a shorter sequence. Again, we have a subtle interaction between specifications: proving the consistency of a pure method makes use of the specification of this method as well as invariants and the specification of the methods mentioned in these invariants.

These examples demonstrate that generating the appropriate proof obligations to check well-formedness is challenging. A useful checker must permit dependencies between specification elements, but prevent circular reasoning.

Approach and contributions. We show well-formedness of specifications by posing proof obligations to ensure: (1) that partial operations are applied within their domains, (2) the existence of a possible result value for each pure method, and (3) that recursive specifications are well-founded. In order to deal with dependencies between pure methods, we determine a dependency graph, which we process bottom-up. Thereby, one can use the properties of a method m to prove the proof obligations for the methods using m.

To deal with partiality, we interpret specifications in 3-valued logic. However, we want to support standard theorem provers, which typically use 2-valued logic and total functions [22, 14]. Therefore, we express the proof obligations in 2-valued logic by applying the Δ formula transformer [17] to the specification expressions. We proved the following soundness result: If all proof obligations for the pure methods of a program are proved then there is a partial model for the axiomatization of these pure methods. In other words, we guarantee that the partiality constraints are satisfied and the axiomatization is consistent.

Our approach differs from existing solutions for theorem provers [11, 22], where consistency is typically enforced by restricting specifications to conservative extensions, but no checks are performed for axioms. Since specifications of pure methods are axiomatic, the approach of conservative extensions is not applicable to contract languages. Moreover, theorem provers require the user to resolve dependencies by ordering the elements of a theory appropriately. We determine this order automatically using a dependency graph.

Our approach improves on existing solutions for program verifiers in three ways. First, it supports (mutually) recursive specifications, whereas in previous work recursive specifications are severely restricted [13, 12]. Second, our approach allows us to use the specification of one method to prove well-formedness of another, which is needed in many practical examples. Such dependencies are not discussed in previous work [9, 13] and are not supported by program verifiers that perform consistency checks, such as Spec#. Neglecting dependencies leads to the rejection of well-formed specifications. Third, we prove consistency for the axiomatization of pure methods; such a proof is either missing in earlier work [9, 12] or only presented for a very restricted setting [13].

For simplicity, we consider pure methods to be strongly-pure. That is, pure methods may not modify the heap in any way. An extension to weakly-pure methods [13], which may allocate and initialize objects, is possible.

Outline. Sec. 2 defines well-formedness of pure-method specifications. We present sufficient proof obligations to guarantee the existence of a model in Sec. 3. We discuss how our technique can be applied with automatic theorem provers in Sec. 4. We summarize related work in Sec. 5 and offer conclusions in Sec. 6.

2 Well-formedness

In this section, we define the well-formedness criteria for the specifications of pure methods. Even though some criteria such as partiality also apply to non-pure methods, we focus on pure methods in the following.

Preliminaries. We assume a set **Heap** of heaps with the usual properties. For simplicity, we assume that a program consists of exactly one class; a generalization to several classes and subclassing is possible.

Since there is a one-to-one mapping between pure methods and the corresponding uninterpreted function symbols, we can state the well-formedness criteria directly on the function symbols. In particular, we say "the specification of a function f" to abbreviate "the specification of the pure method encoded by function f". We assume a signature with the function symbols $\mathbf{F} := \{f_1,\ f_2, , \ldots,\ f_n\}$, which correspond to the pure methods of a program. For simplicity we assume pure methods to have exactly one explicit parameter. Thus, all functions in $\mathbf{F}$ are ternary with parameters for the heap (h), receiver object (o), and explicit parameter (p). We assume that all formulas and terms are well-typed.

We define a specification of $\mathbf{F}$ as $\mathbf{Spec} := \langle \mathbf{Pre}, \mathbf{Post}, \mathbf{INV} \rangle$, where:

- **Pre** maps each $f_i \in \mathbf{F}$ to a formula. We denote $\mathbf{Pre}(f_i)$ as $\mathbf{Pre}_{f_i}$. Due to the syntactic structure of preconditions, the only free variables in $\mathbf{Pre}_{f_i}$ are h, o, and p.
- **Post** maps each $f_i \in \mathbf{F}$ to a formula. We denote $\mathbf{Post}(f_i)$ as $\mathbf{Post}_{f_i}$. Due to the syntactic structure of postconditions, the only free variables in $\mathbf{Post}_{f_i}$ are h, o, p, and the result variable res. Since we assume pure methods to be strongly-pure, one heap variable is enough to capture the heap before and after the method execution.
- **INV** is a set of formulas $\{\mathbf{Inv}_1, \mathbf{Inv}_2, \ldots, \mathbf{Inv}_m\}$. Due to the syntactic structure of invariants, the only free variables in $\mathbf{Inv}_i \in \mathbf{INV}$ are the heap h and the object o to which the invariant is applied.
  We use $\mathbf{SysInv} := \forall\, o \in h.\, \wedge_{i=1}^{m} \mathbf{Inv}_i$ to denote the conjunction of all invariants for all allocated objects, where $o \in h$ expresses that a reference o refers to an allocated object in heap h. Note that **SysInv** is an open formula with free variable h.

Structures and interpretations. To define the interpretation of specifications, we use a structure $\mathbf{M} := \langle \mathbf{Heap}, \mathbf{R}, \mathbf{I} \rangle$, where $\mathbf{R}$ is the set of references and $\mathbf{I}$ is an interpretation function for the specification of a function $f \in \mathbf{F}$: $\mathbf{I}(f) : \mathbf{Heap} \times \mathbf{R} \times \mathbf{R} \to \mathbf{R}$. This structure can be trivially extended to other sorts like integer or boolean.

For a formula φ, we define the interpretation in total structures $[\varphi]^2_{\mathbf{M}} e$ in the standard way. Here, e is a *variable assignment* that maps the free variables of φ to values. For the interpretation in partial structures $[\varphi]^3_{\mathbf{M}} e$, we follow Berezin et al. [5]: intuitively, the interpretation of a function is defined if and only if the interpretations of all parameters are defined and the vector of parameters belongs to the function domain. The interpretation of logical operators and quantifiers is defined according to Kleene logic [20].

A total interpretation maps a formula to a value in $\mathbf{Bool}_2 := \{\mathbf{T}, \mathbf{F}\}$, while a partial interpretation maps a formula to a value in $\mathbf{Bool}_3 := \{\mathbf{T}, \mathbf{F}, \bot\}$. A partial structure $\mathbf{M}$ can be extended to a total structure $\hat{\mathbf{M}}$ by defining values of functions outside of their domains by arbitrary values. To check whether or not a value in $\mathbf{Bool}_3$ is $\bot$ we use the following function:

$$\mathbf{wd} : \mathbf{Bool}_3 \to \mathbf{Bool}_2$$

$$\mathbf{wd}(x) := \begin{cases} \mathbf{T}\ , \text{ if } \ x \in \{\mathbf{T}, \mathbf{F}\} \\ \mathbf{F}\ , \text{ if } \ x = \bot \end{cases}$$

Well-formedness criteria. A specification **Spec** is well-formed (denoted by $\models \mathbf{Spec}$) if there exists a partial model $\mathbf{M}$ for the specification. A structure $\mathbf{M}$ is a *partial model* for specification **Spec**, denoted by $\mathbf{M} \models \mathbf{Spec}$, if it satisfies the following four criteria:

1. Invariants are never interpreted as $\bot$, that is, for each $\mathbf{heap} \in \mathbf{Heap}$:
   $\mathbf{wd}([\mathbf{SysInv}]^3_M e)$ holds
   where $e := [h \to \mathbf{heap}]$.
2. Preconditions are never interpreted as $\bot$ in heaps that satisfy the invariants of all allocated objects, that is, for each $f \in \mathbf{F}$, $\mathbf{heap} \in \mathbf{Heap}$, $\mathbf{this} \in \mathbf{heap}$, and $\mathbf{par} \in \mathbf{heap}$:
   if $[\mathbf{SysInv}]^3_M e$ holds, then $\mathbf{wd}([\mathbf{Pre}_f]^3_{\mathbf{M}} e)$ holds
   where $e := [h \to \mathbf{heap}, o \to \mathbf{this}, p \to \mathbf{par}]$.
3. The values of the parameters belong to the domain of the interpretation of function symbols, provided that the heap satisfies the invariants and the precondition holds. That is, for each $f \in \mathbf{F}$, $\mathbf{heap} \in \mathbf{Heap}$, $\mathbf{this} \in \mathbf{heap}$, and $\mathbf{par} \in \mathbf{heap}$:
   if $[\mathbf{SysInv}]^3_M e$ and $[\mathbf{Pre}_f]^3_{\mathbf{M}} e$ hold,
   then $\langle \mathbf{heap}, \mathbf{this}, \mathbf{par} \rangle \in \mathbf{dom}(\mathbf{I}(f))$ holds
   where $e := [h \to \mathbf{heap}, o \to \mathbf{this}, p \to \mathbf{par}]$.
4. Postconditions are never interpreted as $\bot$ for any result, and the interpretation of function f as result value satisfies the postcondition, provided that the heap satisfies the invariants and the precondition holds. That is, for each $f \in \mathbf{F}$, $\mathbf{heap} \in \mathbf{Heap}$, $\mathbf{this} \in \mathbf{heap}$, and $\mathbf{par} \in \mathbf{heap}$:
   if $[\mathbf{SysInv}]^3_M e$ and $[\mathbf{Pre}_f]^3_{\mathbf{M}} e$ hold,
   then for each $\mathbf{result} \in \mathbf{heap}$ $\mathbf{wd}([\mathbf{Post}_f]^3_{\mathbf{M}} e')$ holds,
   and $[\mathbf{Post}_f]^3_{\mathbf{M}} e$ holds
   where $e := [h \to \mathbf{heap}, o \to \mathbf{this}, p \to \mathbf{par}, res \to \mathbf{I}(f)(\mathbf{heap}, \mathbf{this}, \mathbf{par})]$,
   $e' := [h \to \mathbf{heap}, o \to \mathbf{this}, p \to \mathbf{par}, res \to \mathbf{result}]$.

Axiomatization. As motivated in Sec. 1, a verification system needs to extract axioms from the specifications of pure methods. We denote the axiom for function symbol f as $\mathbf{Ax}_f$ and the axioms for all functions as $\mathbf{Ax}_{\mathbf{Spec}}$. Formally:

$$\mathbf{Ax}_f := \forall\, h, o \in h, p \in h.\ \mathbf{SysInv} \wedge \mathbf{Pre}_f \Rightarrow \mathbf{Post}_f[f(h,o,p)/res]$$

$$\mathbf{Ax}_{\mathbf{Spec}} := \bigwedge_{f \in \mathbf{F}} \mathbf{Ax}_f$$

From well-formedness criterion 4 and $\mathbf{Ax}_f$, we can conclude that if a structure $\mathbf{M}$ is a partial model for specification **Spec** then it is a model for $\mathbf{Ax}_{\mathbf{Spec}}$:

$$\text{if} \quad \mathbf{M} \models \mathbf{Spec} \quad \text{then} \quad \mathbf{M} \models \mathbf{Ax}_{\mathbf{Spec}}$$

Consequently, if specification **Spec** is well-formed then the axioms are consistent:

$$\text{if} \quad \models \mathbf{Spec} \quad \text{then} \quad \models \mathbf{Ax}_{\mathbf{Spec}}$$

Important to note is that this property does not hold in the other direction, that is, if $\models \mathbf{Ax}_{\mathbf{Spec}}$ then $\models \mathbf{Spec}$ is not necessarily true. For example, consider a method with precondition `1/0 == 1/0` and postcondition `true`. In 2-valued logic, the axiom is trivially consistent, but the specification is not well-formed (criterion 2). This demonstrates that our well-formedness criteria require more than just consistency, namely also satisfaction of partiality constraints.

3 Checking well-formedness

In this section, we present sufficient proof obligations that ensure that a specification is well-formed, that is, the existence of a model.

3.1 Partiality

We want our technique to work with first-order logic theorem provers, which are often used in program verifiers. These provers check that a formula holds for all total models. However, we need to check properties of partial models. Therefore, we apply a technique that reduces the 3-valued domain to a 2-valued domain by ensuring that $\perp$ is never encountered. This is a standard technique applied in different tools, for instance, in B [4], CVC Lite [5], and ESC/Java2 [9].

The main idea is to use the formula transformer Δ [17, 4], which takes a (possibly open) formula φ and *domain restriction* δ, and produces a new formula φ'. The interpretation of φ' in 2-valued logic is true if and only if the interpretation of φ in 3-valued logic is different from $\perp$. The domain restriction δ is a mapping from a set of function symbols $\mathbf{F}_\delta$ to formulas. δ characterizes the domains of the function symbols of $\mathbf{F}_\delta$. For instance for the division operator, the domain restriction δ requires the divisor to be non-zero. Thus, $\Delta(a/b > 0, \delta) \equiv b \neq 0$.

For lack of space, we do not give the details of the Δ operator and refer the reader to [4]. The most important property for our purpose is the following [5]:

$$\mathbf{M} \models \delta \;\Rightarrow\; (\, [\Delta(\varphi, \delta)]^2_{\hat{M}} e = \mathbf{wd}([\varphi]^3_M e)\,) \tag{1}$$

which captures the intuition of Δ described above. Δ is a syntactical characterization of the semantical operation **wd**. Thus, using Δ, we can check in 2-valued logic the partiality properties we are interested in.

Property (1) interprets formulas w.r.t. a structure $\mathbf{M}$. This structure with function symbols $\mathbf{F}_\delta$ has to be a model for δ (denoted by $\mathbf{M} \models \delta$), that is:

- The domain formulas are defined, that is, for each $f \in \mathbf{F}_\delta$

  $\mathbf{wd}([\delta(f)]^3_M e)$ holds for all e.
- δ characterizes the domains of function interpretations for $\mathbf{M}$, that is, for each $f \in \mathbf{F}_\delta$ and $\mathbf{val}_1, \ldots, \mathbf{val}_k \in \mathbf{R}$:

  $[\delta(f)]^3_{\mathbf{M}} e$ holds if and only if $\langle \mathbf{val}_1, \ldots, \mathbf{val}_k \rangle \in \mathbf{dom}(\mathbf{I}(f))$

  where $e := [v_1 \to \mathbf{val}_1, \ldots, v_k \to \mathbf{val}_k]$ and $\{v_1, \ldots, v_k\}$ are the parameter names of f. (Since methods have only one explicit parameter, $k = 3$.)

3.2 Incremental construction of model

In general, showing the existence of a model requires one to prove the existence of all its functions. To be able to work with first-order logic theorem provers, we approximate this second-order property in first-order logic. We generate proof obligations whose validity in 2-valued first-order logic guarantees the existence

of a model. However, if we fail to prove them then we do not know whether a model exists or not. That is, the procedure is sound but not complete. However, it works for the practical examples we have considered so far.

The basic idea of our procedure is to construct a model incrementally. We build a dependency graph whose nodes are function symbols and invariants. There is an edge from node a to node b if the specification of function a or the invariant a applies function b. The dependency graph of interface `Sequence` is presented in Fig. 2.

The dependency graph may be cyclic. However, we disallow cycles that are introduced by preconditions. In other words, a precondition must not be recursive in order to avoid fix-point computation to define the domain of the function. This is not a limitation for practical examples.

We construct the model by traversing the dependency graph bottom-up. We start with the empty specification $\mathbf{Spec}_0 := \langle \emptyset, \emptyset, \emptyset \rangle$, for which we trivially have a model $\mathbf{M}_0$. In each step j, we select a set of nodes $G_j := \{g_1, g_2, \ldots, g_k\}$ such that if there is an edge from g_i to a node n then either n has already been visited in some previous step (i.e., $n \in G_1 \cup ... \cup G_{j-1}$) or $n \in G_j$. Moreover, we choose G_j such that it has one of the following forms:

1. G_j contains exactly one invariant $\mathbf{Inv}_l \in \mathbf{INV}$.
2. G_j contains exactly one function symbol $f_l \in \mathbf{F}$ and the specification of f_l is not recursive.
3. G_j is a set of function symbols, and the nodes in G_j form a cycle in the dependency graph, that is, they are specified recursively in terms of each other. G_j may contain only one node in case of direct recursion.

We call the pre- and postconditions and the invariants of G_j the *current specification fragment*, s_j. We extend $\mathbf{Spec}_{j-1}$ with s_j resulting in $\mathbf{Spec}_j$. We impose proof obligations on s_j that guarantee that the model $\mathbf{M}_{j-1}$ for $\mathbf{Spec}_{j-1}$ can be extended to a model $\mathbf{M}_j$ for $\mathbf{Spec}_j$. Since this construction is inductive, we may assume that all specification fragments processed up to step $j-1$ are well-formed.

It is easy to see that an order in which one can traverse the dependency graph always exists. However, the chosen order may influence the success of the model construction. Essentially one should choose an invariant node whenever possible because the invariant provides information that might be useful for later steps.

3.3 Proof obligations

We now present the proof obligations for the three different kinds of current specification fragments s_j. We refer to the elements of $\mathbf{Spec}_j$ as $\mathbf{Pre}_j$, $\mathbf{Post}_j$, and $\mathbf{INV}_j$. To make the formulas more readable we use the following notations:

- $\mathbf{SysInv}_j := \forall\ o \in h.\ \bigwedge_{Inv \in \mathbf{INV}_j} Inv.\ o$. $\mathbf{SysInv}_j$ is the conjunction of invariants processed up to step j. After the last step z of the construction of the model, we have $\mathbf{SysInv}_z = \mathbf{SysInv}$.
- F_j denotes the set of function symbols whose pre- and postconditions have been processed up to step j: $F_j := \mathbf{dom}(\mathbf{Pre}_j)$, and thus $F_j = \mathbf{dom}(\mathbf{Post}_j)$.

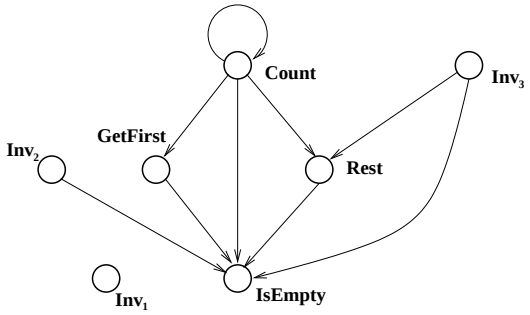

Fig. 2. Dependency graph for interface `Sequence`.

– We denote the axioms for $\mathbf{Spec}_j$ as follows:

$$\mathbf{Ax}_f^j := \forall\ h, o \in h, p \in h.\ \mathbf{SysInv}_j \wedge \mathbf{Pre}_f \Rightarrow \mathbf{Post}_f[f(h,o,p)/res]$$

$$\mathbf{Ax}_{\mathbf{Spec}_j} := \bigwedge_{f \in F_j} \mathbf{Ax}_f^j$$

$\mathbf{Ax}_f^j$ is the definition of the axiom for a function f according to specification $\mathbf{Spec}_j$. Note that the axiom $\mathbf{Ax}_f^j$ may be different for different j since $\mathbf{SysInv}_j$ gets gradually strengthened during the construction of the model. Therefore, the axiom $\mathbf{Ax}_f^j$ becomes gradually weaker. This is an important observation for the soundness of our approach. After the last step z of the construction of the model, we have $\mathbf{Ax}_f^z = \mathbf{Ax}_f$ and $\mathbf{Ax}_{\mathbf{Spec}_z} = \mathbf{Ax}_{\mathbf{Spec}}$.

The following proof obligations are posed on the three different types of specification fragments in step j.

Invariant $\mathbf{Inv}_l$. The invariant $\mathbf{Inv}_l$ must be well-defined for each object, provided the invariants $\mathbf{SysInv}_{j-1}$ hold.

$$\mathbf{Ax}_{\mathbf{Spec}_{j-1}} \Rightarrow \forall\ h.\ (\mathbf{SysInv}_{j-1} \Rightarrow \Delta(\forall\ o \in h.\ \mathbf{Inv}_l, \mathbf{Pre}_{j-1})) \tag{2}$$

Note that we use preconditions $\mathbf{Pre}_{j-1}$ as domain restriction. Although invariants additionally restrict the domain of functions, these restrictions are never violated due to the assumption that $\mathbf{SysInv}_{j-1}$ holds.

Example. We instantiate the proof obligation for a specification fragment from Fig. 1. The corresponding dependency graph is presented in Fig. 2. The traversal of the dependency graph first visits the first invariant since it has no dependencies. The well-definedness of the invariant is trivial. Next, the traversal takes method `IsEmpty`, which is also processed trivially since the method has no specifications. As third node, the second invariant is picked. For this specification fragment, the following proof obligation is generated.

$$\forall\ h.\ ((\forall\ o \in h.\ h[o, Length] \geq 0) \Rightarrow \\ \Delta(\forall\ o \in h.\ IsEmpty(h,o) \Rightarrow h[o, Length] = 0, \{\langle IsEmpty, true\rangle\}))$$

where $h[o, f]$ denotes field access with receiver object o and field f in heap h. Note that $\mathbf{Ax_{Spec}}_2$ has been omitted since it is equivalent to true. After the application of the Δ operator, the proof obligation requires one to prove that (1) o is non-null since it is the receiver of a method call and a field access, and that (2) the domain restriction of *IsEmpty* is not violated. The first property holds since $o \in h$, the second since the domain restriction of *IsEmpty* is true. □

Pre- and postcondition of a single function f_l. This case requires two proof obligations for the non-recursive pre- and postcondition of f_l, respectively. The first proof obligation checks that the precondition of f_l is defined for all receiver objects and parameters in all heaps in which the invariants hold.

$$\mathbf{Ax_{Spec}}_{j-1} \Rightarrow \forall\ h, o \in h, p \in h.\ (\mathbf{SysInv}_{j-1} \Rightarrow \Delta(\mathbf{Pre}_{f_l}, \mathbf{Pre}_{j-1})) \tag{3}$$

Example. Assume method `Rest` is selected as fourth specification fragment. The corresponding proof obligation is the following.

$$\begin{array}{l}\forall\ h, o \in h. \\ \quad (\ (\forall\ o \in h.\ h[o, Length] \geq 0\ \wedge\ (IsEmpty(h, o) \Rightarrow h[o, Length] = 0)) \Rightarrow \\ \quad\ \Delta(\neg IsEmpty(h, o), \{\langle IsEmpty, true\rangle\})\ )\end{array}$$

Again, $\mathbf{Ax_{Spec}}_3$ has been omitted since it is equivalent to true. After the application of the Δ operator, the same properties need to be proven as above: o is non-null and the domain restriction of *IsEmpty* is not violated. □

The second proof obligation checks that the postcondition of f_l is never interpreted as $\perp$ for any result, and that there exists a value which satisfies the postcondition for all receiver objects and parameters that satisfy the precondition in all heaps in which the invariants hold.

$$\begin{array}{l}\mathbf{Ax_{Spec}}_{j-1} \Rightarrow \forall\ h, o \in h, p \in h.\ (\ \mathbf{SysInv}_{j-1}\ \wedge\ \mathbf{Pre}_{f_l}\ \Rightarrow \\ \qquad (\forall\ res.\ \Delta(\mathbf{Post}_{f_l}, \mathbf{Pre}_{j-1})) \wedge (\exists\ res.\ \mathbf{Post}_{f_l})\ )\end{array} \tag{4}$$

Example. The proof obligation for the postcondition of method `Rest` is:

$$\begin{array}{l}\forall\ h, o \in h. \\ \quad (\ (\forall\ o \in h.\ h[o, Length] \geq 0\ \wedge\ (IsEmpty(h, o) \Rightarrow h[o, Length] = 0)) \wedge \\ \quad\ \neg IsEmpty(h, o) \\ \qquad \Rightarrow \\ \quad\ (\forall\ res.\ \Delta(res \neq null, \{\langle IsEmpty, true\rangle\})) \wedge (\exists\ res.\ res \neq null)\ )\end{array}$$

As before, $\mathbf{Ax_{Spec}}_3$ is equivalent to true. The first conjunct is proved trivially since formula $res \neq null$ does not contain any partial operation. To satisfy the second conjunct, we instantiate res with o. □

Pre- and postconditions of a set of recursively-specified functions. This case handles both direct and mutual recursion. That is, we have a set of functions $G_j := \{g_1, g_2, \ldots, g_k\}$ with $k \geq 1$. We assume that for each function g_i in G_j the programmer provides a measure function $\|\cdot\|_{g_i} : \mathbf{Heap} \times \mathbf{R} \times \mathbf{R} \to \mathbb{N}$ using the `Measure` attribute. We assume that there is no recursion via measure functions, that is, the definition of measure function $\|\cdot\|_{g_i}$ may only contain function symbols from $G_1 \cup \ldots \cup G_{j-1}$, but not from G_j.

Since preconditions must not be recursively specified (see Sec. 3.2), the proof obligation for the precondition of each g_i is identical to proof obligation (3) for the non-recursive case.

In order to prove well-formedness of postconditions, we first need to show that user-specified measures are well-defined and non-negative. For a function g_i with measure attribute `Measure=`μ_{g_i}, we introduce a new pure method M_{g_i} with precondition $\mathbf{Pre}_{g_i}$ and postcondition $\mu_{g_i} \geq 0$. The dependency graph is extended with a node for M_{g_i} and an edge from g_i to M_{g_i}. Node M_{g_i} is processed like any other node. This allows measures to rely on invariants and to contain calls to pure methods.

Proof obligation (5) below for postconditions is similar to proof obligation (4), but differs in two ways: First, we have to prove that the recursive specification is well-founded. Since we have already shown that our measure functions yield non-negative numbers, it suffices to show that the measure decreases for each recursive application. We achieve this by using a domain restriction that additionally requires the measure for recursive applications to be lower than the measure *ind* of the function being specified. If the measure *ind* is 0, the domain restriction becomes false, which prevents further recursion. Note that the occurrence of *ind* seems to violate the condition that domain restrictions do not contain free variables other than the parameters of the function whose domain they characterize. However, since *ind* is universally quantified, we may consider *ind* to be a constant for each particular application of the domain restriction. (One could think of the universal quantification as an unbounded conjunction, where *ind* is a constant in each of the conjuncts.)

Second, for the proof of well-formedness of the specification of a function g_i, we may assume the properties of the functions recursively applied in this specification. This is an induction scheme over the measure *ind*, which is expressed by the assumption in lines 4 and 5 of the following proof obligation, which must be shown for each method g_i.

$$
\begin{array}{l}
\mathbf{Ax}_{\mathbf{Spec}_{j-1}} \Rightarrow \\
\quad \forall\, ind \in \mathbb{N}, h, o \in h, p \in h. \\
\quad\quad (\, \mathbf{SysInv}_{j-1} \;\wedge\; \mathbf{Pre}_{g_i} \;\wedge\; \|\langle h, o, p\rangle\|_{g_i} = ind \;\wedge \\
\quad\quad\quad (\, \bigwedge_{l=1}^{k} \forall\, o' \in h, p' \in h.\; \mathbf{Pre}_{g_l}[o'/o, p'/p] \;\wedge\; \|\langle h, o', p'\rangle\|_{g_l} < ind \Rightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{Post}_{g_l}[o'/o, p'/p, g_l(h, o', p')/res]\,) \\
\quad\quad\quad \Rightarrow \\
\quad\quad\quad (\forall\, res.\; \Delta(\mathbf{Post}_{g_i}, \mathbf{Pre}_{j-1} \cup \{\langle g_l, \mathbf{Pre}_{g_l} \wedge \|\langle h, o, p\rangle\|_{g_l} < ind\rangle \mid l \in 1..k\})) \;\wedge \\
\quad\quad\quad (\exists\, res.\; \mathbf{Post}_{g_i})\,)
\end{array}
\tag{5}
$$

Example. Since the size of proof obligation (5) for the postcondition of method `Count` (the only recursive specification in our example) is rather large, we use a considerably smaller example here, namely the factorial function with the following specification.

```
[Pure][Measure=p] int Fact(int p)
  requires p >= 0;
  ensures p == 0 ==> result == 1;
  ensures p > 0 ==> result == Fact(p-1)*p;
```

To simplify the example, we omit the variables for heap h and receiver object o.

First, we need to prove that measure p is well-defined and non-negative. This is trivially proven since the measure does not contain partial operators and the precondition of `Fact` guarantees that p is non-negative.

Next, we need to show proof obligation (5). For brevity, we only show it for the second postcondition, which is the interesting case containing recursion:

$$
\begin{array}{l}
\forall\, ind \in \mathbb{N}, p. \\
\quad (\, p \geq 0 \ \wedge\ p = ind \wedge \\
\quad\ (\, \forall\, p'.\ p' \geq 0 \wedge p' < ind \ \Rightarrow \\
\qquad\quad (p' = 0 \Rightarrow Fact(p') = 1\,) \ \wedge\ (p' > 0 \Rightarrow Fact(p') = Fact(p'-1) * p')\,) \\
\qquad \Rightarrow \\
\quad\ (\forall\, res.\ \Delta(\, p > 0 \Rightarrow res = Fact(p-1) * p, \{\langle Fact,\ p \geq 0 \wedge p < ind \rangle\ \})) \wedge \\
\quad\ (\exists\, res.\ p > 0 \Rightarrow res = Fact(p-1) * p)\,)
\end{array}
$$

We need to show that the two quantified conjuncts on the right-hand side of the implication hold. Proving that the existential holds is straightforward due to the equality. The other conjunct is more interesting. The only partial operator is $Fact$ and after applying the Δ operator the sub-formula simplifies to:

$$\forall\, res.\ p > 0 \Rightarrow p - 1 \geq 0 \ \wedge\ p - 1 < ind$$

The first conjunct is provable from $p > 0$ and the second from $p = ind$ in the premise of the proof obligation. □

Soundness. The above proof obligations are sufficient to show that a specification is well-formed:

> **Theorem.** If a specification **Spec** does not contain recursive preconditions and all of the above proof obligations for **Spec** hold then **Spec** is well-formed, that is, $\models$ **Spec** holds.

The proof of this theorem runs by induction on the order of specification fragments given by the dependency graph. For each recursive specification fragment, the proof uses a nested induction on the recursion depth *ind*. Due to lack of space, we refer to [23] for a detailed proof sketch.

Modularity. In general, adding new classes to a program does not invalidate the proofs for the well-formedness criteria of existing methods and invariants. This is because we assume behavioral subtyping, which ensures that the axiom for an overriding method is weaker than the axiom for the overridden method. Although new classes can introduce cycles in the dependency graph that involve existing methods, proofs remain valid since we introduce new function symbols for overriding methods, which thus do not interfere with existing proofs.

The invariants of additional classes strengthen **SysInv**, which appears as part of the premises of proof obligations; thus, they weaken the proof obligations.

4 Application with automatic theorem provers

The proof obligations presented in the previous section are sufficient to show the well-formedness of a specification. However, they are not well-suited for automatic theorem provers such as Simplify [14] or Z3 for two reasons. First, the proof obligations to ensure consistency for postconditions (proof obligations (4) and (5)) contain existential quantifiers, for which automatic theorem provers often do not find suitable instantiations. Second, the proof obligation for the well-foundedness of recursive specifications (proof obligation (5)) is in general proved by induction on *ind*, but induction is not supported well by automatic theorem provers. In this section, we discuss these issues.

Consistency. Spec# uses four approaches to find witnesses for the satisfiability of a specification, that is, instantiations for the existential quantifiers[1]. First, if a postcondition has the form `result R E`, where `R` is a reflexive operator and `E` is an expression that does not contain `result` and recursive calls, then there always exists a possible result value, namely, the value of `E` [12]. Thus, this part of the proof obligations can be dropped. Second, if a pure method has a body of the form `return E`, where `E` does not contain a recursive call, then expression `E` is a likely candidate for a witness. It suffices to use a simplified proof obligation to show that this candidate actually is a witness. Third, for many postconditions, good candidates for witnesses can be inferred by simple heuristics. For instance, for a postcondition `result > E`, one might try `E + 1`. Finally, if the former approaches do not work, Spec# allows programmers to specify witnesses for model fields explicitly. One could use the same approach for pure methods.

Well-foundedness. Proof obligation (5) in general requires induction. For instance, if function $f(n)$ has a postcondition $(n = 0 \Rightarrow res = 1) \wedge (n > 0 \Rightarrow res = 1/f(n-1))$, one needs to apply induction to prove that f never returns zero. However, induction is needed only if the function is specified recursively *and* the recursive call occurs as an argument to a partial function, as in this example. In our experience, this is not the case for most specifications. For instance, proving proof obligation (5) for the factorial function does not require induction, as we have shown in Sec. 3.3. Therefore, this proof obligation is not a major limitation in practice.

[1] Most of these approaches were proposed and implemented by Rustan Leino and Ronald Middelkoop.

5 Related work

We sketch what three important groups of formal systems do in the areas of consistency and well-definedness checking.

Theorem provers. Isabelle [22] is an interactive LCF-style theorem proving framework based on a small logical core. Everything on top of the core is supposed to be defined by conservative extensions, which ensures the consistency of the specification. The use of axioms is possible but discouraged since inconsistency may be introduced. Recursion (both direct and mutual) is supported and the well-foundedness of the recursion has to be proven. Isabelle handles partiality by under-specification [15] and requires no well-definedness checks.

PVS [11] is similar to Isabelle with respect to consistency guarantees. The main difference is in the modeling of partial functions. Although PVS also considers functions to be total, predicate subtyping is used to restrict the domain of functions. This makes the type system undecidable leading to Type Correctness Conditions to be proven [24].

Formal software development systems. Z is a formal specification language for computing systems [25]. The work closest to ours is the approach of Hall et al., which shows how a model conjecture can be derived from a Z specification [16]. Partiality is handled by under-specification [26].

The B method [1] is similar to Z but is more focused on the notion of refinement. Satisfiability of the specification has to be proven in each refinement step. B allows users to add axioms whose consistency is not checked. Thus, they may introduce unsoundness. B allows functions to be partial and requires specifications to be well-defined by using the Δ formula transformer [4].

VDM [18] also checks satisfiability of specifications and allows the use of (possibly inconsistent) axioms. VDM uses LPF [3], a 3-valued logic. In contrast to our approach, well-definedness is not proven before the actual proof process, but is proven together with the validity of verification conditions.

Program verifiers. ESC/Java2 [19] is an automatic extended static checker for Java programs annotated with JML specifications. The tool axiomatizes specifications of pure methods [10]. Consistency of the axiom system is not ensured, which can lead to unsoundness. Recently, well-definedness checks have been added by Chalin [9] but it is not clear how dependencies among specification elements are handled, and no soundness proof is provided.

Jack [7] is a program verifier for JML annotated Java programs. The backend prover of the tool is Coq [6]. The tool axiomatizes pre- and postconditions of pure methods separately. This separation ensures that axioms are only instantiated when a pure-method call occurs in a given verification condition—as opposed to be available to the theorem prover at any time. However, since Jack does not check consistency, unsoundness can still occur by the use of axioms. Jack does not support mutual recursion and does not check well-definedness.

The Spec# program verifier ensures consistency of axioms over pure methods by the approaches described in Sec. 4 and by allowing programmers to declare a static call-order on pure methods. Only a simple form of recursive specifications is supported where the measure is based on the ownership relation. The well-foundedness of this relation can be checked by the compiler without proof obligations [12]. Spec# does not fully check well-definedness of specifications.

Our technique improves on our own earlier work [13] by allowing pure-method calls in invariants, ensuring well-formedness of specifications, supporting mutual recursion, taking dependencies into account, and by precisely defining what the proposed proof obligations guarantee. On the other hand, [13] handles weak-purity which we omitted in this paper for simplicity. However, our work could be extended following the technique described in [13].

6 Conclusion

Well-formedness of specifications is important to meet programmer expectations, to reconcile static and runtime assertion checking, and to ensure soundness of static verification. We presented a new technique to check the well-formedness of specifications. We showed how to incrementally construct a model for the specification, which guarantees that the partiality constraints of operations are respected and that the axiomatization of pure methods is consistent. Our technique can be applied in any verification system, regardless of its contract language, logic, or backend theorem prover. As a proof of concept, we implemented our technique in the Spec# verification system.

As future work, we plan to develop adapted proof obligations that require induction in fewer cases. We expect that this can be done by generating specific proof obligations for each given recursive call, which encode the inductive argument. We also plan to investigate how to conveniently specify measures for methods that traverse object structures.

Acknowledgments. We are grateful to Julien Charles, Farhad Mehta, and Burkhart Wolff for helpful discussions on related work. Thanks also to the anonymous reviewers for their insightful comments. Geraldine von Roten implemented the presented technique in the Spec# system.

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

1. J. R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.

3. H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.
4. P. Behm, L. Burdy, and J.-M. Meynadier. Well Defined B. In *International B Conference*, pages 29–45. Springer-Verlag, 1998.
5. S. Berezin, C. Barrett, I. Shikanian, M. Chechik, A. Gurfinkel, and D. L. Dill. A practical approach to partial functions in CVC Lite. In *PDPAR*, 2004.
6. Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.
7. L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In *FME*, volume 2805 of *LNCS*, pages 422–439. Springer-Verlag, 2003.
8. P. Chalin. Are the logical foundations of verifying compiler prototypes matching user expectations? *Formal Aspects of Computing*, 19(2):139–158, 2007.
9. P. Chalin. A sound assertion semantics for the dependable systems evolution verifying compiler. In *ICSE*, pages 23–33. IEEE Computer Society, 2007.
10. D. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, October 2005.
11. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. *A Tutorial Introduction to PVS*, April 1995.
12. Á. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, volume 4422 of *LNCS*, pages 336–351. Springer-Verlag, 2007.
13. Á. Darvas and P. Müller. Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology (JOT)*, 5(5):59–85, June 2006.
14. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.
15. D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *LNCS*, pages 366–373. Springer-Verlag, 1995.
16. J. G. Hall, J. A. McDermid, and I. Toyn. Model conjectures for Z specifications. In *7th International Conference on Putting into Practice Methods and Tools for Information System Design*, pages 41–51, 1995.
17. A. Hoogewijs. On a formalization of the non-definedness notion. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 25:213–217, 1979.
18. C. B. Jones. *Systematic software development using VDM*. Prentice Hall, 1986.
19. J. R. Kiniry and D. R. Cok. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS*, volume 3362 of *LNCS*, pages 108–128. Springer-Verlag, 2005.
20. S. C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.
21. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
22. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
23. A. Rudich, Á. Darvas, and P. Müller. Checking well-formedness of pure-method specifications (Full Paper). Technical Report 588, ETH Zurich, 2008.
24. J. Rushby, S. Owre, and N. Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.
25. J. M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, 1988.
26. S. H. Valentine. Inconsistency and Undefinedness in Z - A Practical Guide. In *International Conference of Z Users*, pages 233–249. Springer-Verlag, 1998.