Proving Consistency of Pure Methods and Model Fields

K. Rustan M. Leino[1] and Ronald Middelkoop[2]

[1] Microsoft Research, Redmond, USA. `leino@microsoft.com`
[2] Technische Universiteit Eindhoven, Holland. `r.middelkoop@tue.nl`

Abstract. Pure methods and model fields are useful and common specification constructs that can be interpreted by the introduction of axioms in a program verifier's underlying proof system. Care has to be taken that these axioms do not introduce an inconsistency into the proof system. This paper describes and proves sound an approach that ensures no inconsistencies are introduced. Unlike some previous syntax-based approaches, this approach is based on semantics, which lets it admit some natural but previously problematical specifications. The semantic conditions are discharged by the program verifier using an SMT solver, and the paper describes heuristics that help avoid common problems in finding witnesses with trigger-based SMT solvers. The paper reports on the positive experience with using this approach in Spec# for over a year.

1 Introduction

Pure methods and model fields [1, 2] are useful and common specification constructs. By marking a method as pure, the specifier indicates that it can be treated as a function of the state. It can then be called in specifications. Model fields provide a way to abstract from an object's concrete data. A problem with either technique is that it can introduce an inconsistency into the underlying proof system. In this paper, we discuss how to prove (automatically) that no such inconsistency is introduced while allowing a rich set of specifications.

Starting from a review of the setting, the problem, and previous solutions, this section leads up to an overview of our contributions.

Pure Method Specifications. Figure 1 shows the template for a pure method specification (for simplicity, we show only a single formal parameter, named `p`). As usual, **requires** declares the method's precondition P, **ensures** declares the method's postcondition Q, and **result** denotes the method's return value. The only free variables allowed in P are `this` and `p`. In Q, **result** is allowed as well.

A Deduction System. Marking method m as pure adds an uninterpreted total function $\#m : C \times T' \to T$ (a *method function* [3]) to the specification language. In predicates in the specification, the expression $E_0.m(E_1)$ is treated as syntactic

```
pure T m(T' p)
  requires P;
  ensures Q;
```

Fig. 1. Template

```
pure int bad()
  ensures false;
  { return 4; }
```

Fig. 2. Inconsistency

```
pure int n(int i)
  ensures result = this.p(i);
pure int p(int i)
  ensures result = this.n(i)+1;
```

Fig. 3. Harmful indirect recursion

```
class Node {
  Object val;
  rep Node next;

  pure int count(Object obj)
    ensures result = (obj = this.val ? 1 : 0) +
                     (this.next = null ? 0 : this.next.count(obj));

  pure bool has(Object obj)
    ensures result = this.count(obj) > 0;
} //rest of class omitted
```

Fig. 4. Singly linked list (see Sect. 4.1 for **rep**)

sugar for $\#m(E_0, E_1)$. Furthermore, method function $\#m$ is axiomatized in the underlying deduction system for first-order logic by the following axiom:[1]

$$\forall \sigma \in \Sigma \bullet \ [\![\forall \texttt{this}\colon C,\ \texttt{p}\colon T' \bullet \quad P \Rightarrow Q[\#m(\texttt{this},\texttt{p})/\textbf{result}]\ ]\!]\sigma \qquad (1)$$

Here, Σ denotes the set of well-formed program states. Partial function $[\![E]\!]\sigma$ evaluates expression E to its value in state σ. $[\![\#m(E_0, E_1)]\!]\sigma$ is defined as $\#m(\ [\![E_0]\!]\sigma,\ [\![E_1]\!]\sigma)$. Other details of this evaluation are unimportant here. $P[E/v]$ denotes the predicate like P, but with capture-avoiding substitution of variable v by E. For instance, pure method `has` from Fig. 4 introduces uninterpreted total function $\#\texttt{has} : \texttt{Node} \times \texttt{Object} \rightarrow \textbf{bool}$, and axiom $\forall \sigma \in \Sigma \bullet [\![\forall \texttt{this}\colon \texttt{Node},\ \texttt{obj}\colon \textbf{bool} \bullet \#\texttt{has}(\texttt{this},\texttt{obj}) = \#\texttt{count}(\texttt{this},\texttt{obj}) > 0]\!]\sigma$.

Consistency of Deduction System. If one is not careful, pure methods can introduce an inconsistency into the deduction system. As an obvious example, consider Fig. 2. This definition introduces *false* as an axiom into the deduction system (more precisely, it introduces $\forall \sigma \in \Sigma \bullet\ [\![\forall \texttt{this}\colon C \bullet \mathit{false}]\!]\sigma$). So, it has to be ensured that for all possible values of the arguments of method function $\#m$, there is a value that the function can take. Insuring this by requiring a proof of total correctness of the implementation of m before adding the axiom is highly impractical. If $\#m$ is constrained only by the axiom introduced by m, then it suffices to prove property (2):

$$\forall \sigma \in \Sigma \bullet\ [\![\forall \texttt{this}\colon C,\ \texttt{p}\colon T' \bullet \exists x\colon T \bullet \quad P \Rightarrow Q[x/\textbf{result}]\ ]\!]\sigma \qquad (2)$$

If other axioms can also constrain $\#m$, as is the case in the presence of mutual recursion, then property (2) needs to simultaneously mention all methods

[1] The axiomatization differs slightly in the presence of class invariants. To simplify the presentation, invariants are not considered.

```
pure int findInsertionPosition(int N)      pure int max(int x, int y)
  requires 0 ≤ N;                            ensures (x ≤ y ⇒ result = y) ∧
  ensures 0 ≤ result ∧ result ≤ N;          (y ≤ x ⇒ result = x);
```

Fig. 5. Previous syntactic checks forbid these methods; our semantic checks allow them.

```
pure bool isEven(int n)                      pure bool isOdd(int m)
  requires 0 ≤ n;                              requires 0 ≤ m;
  ensures result =                             ensures result ≠ this.isEven(m);
    (n = 0 ? true : this.isOdd(n-1));          measuredBy 2m+1;
  measuredBy 2n;
```

Fig. 6. Odd and even (see Sect. 4.3 for **measuredBy**)

involved. We aim for sound *modular verification*, which means being able to verify a program's modules separately, just like a compiler performs separate compilation of modules. If the mutual recursion can span module boundaries, then there may be no verification scope that has information about all the methods that need to be simultaneously mentioned. Therefore, the consistency of mutual recursion among pure methods is usually stated in a form different from (2).

Previous Solutions. Darvas and Müller [4] prove that inconsistency is prevented if the following two measures are taken: (A) the axiom that is introduced into the deduction system for a method function $\#m$ is not proposition (1), but (2) $\Rightarrow$ (1), and (B) recursion in the pure method axioms is disallowed unless it is direct and well-founded. For example, measure A prevents the pure methods in Fig. 2 from introducing an inconsistency, and measure B forbids the specifications in Fig. 3, whose axioms would otherwise introduce an inconsistency.

Darvas and Leino [3] discuss a problem with measure A, namely that an axiom of the form (2) $\Rightarrow$ (1) is not suitable for automatic reasoning using today's trigger-based SMT solvers like Simplify and Z3 [5, 6]. More specifically, these solvers are unable to come up with a witness for the existential quantification in (2) even in simple cases. This means that property (1) is 'silently ignored', which renders the pure method useless (and possibly confuses the user).

To circumvent the practical problem with measure A, Darvas and Leino introduce a simple syntactic check that allows one to conclude that (2) holds once and for all [3]. Thus, (1) can be introduced as an axiom into the deduction system without fear of inconsistencies. However, the syntactic check is restrictive and prevents a number of natural and useful specifications, including the two in Fig. 5. Syntactic checks cannot guarantee the consistency of `findInsertionPosition`, because its result value is constrained by two inequalities, or of `max`, because its result-value constraints are guarded by antecedents.

Measure B is a Draconian way of dealing with mutual recursion. The syntactic check of Darvas and Leino [3] improves on this situation. However, this check is still restrictive; for instance, it does not permit the example in Fig. 6.

```
class Rectangle {
 int x1,y1,x2,y2; //lower left and upper right corner
 model int width satisfies this.width = this.x2-this.x1;
 model int height satisfies this.height = this.y2-this.y1;
 void scaleH(int factor)
  requires 0 ≤ factor;
  ensures this.width = old(this.width) ∗ factor/100;
  { this.x2 := (this.x2 - this.x1 ) ∗ factor/100 + this.x1; }
} //rest of class omitted
```

Fig. 7. Model fields

A Glimpse of Our Semantic Solution. In our solution, we use heuristics to guess candidate witness expressions for (2). Then we verify that in every program state allowed by the pure method's precondition, one of these candidates establishes the postcondition. For example, for pure method `max` in Fig. 5, we generate three candidate witnesses 1, x, and y, and construct a program snippet of the form:

$$\begin{array}{ll} r := 1; & \textbf{if } ((x \le y \Rightarrow r = y) \wedge (y \le x \Rightarrow r = x)) \; \{ \; \textbf{return } r; \; \} \\ r := x; & \textbf{if } ((x \le y \Rightarrow r = y) \wedge (y \le x \Rightarrow r = x)) \; \{ \; \textbf{return } r; \; \} \\ r := y; & \textbf{return } r; \end{array}$$

and then attempt to verify, using our program verifier's machinery, that this program snippet establishes the postcondition of the pure method.

Model Fields. Model fields introduce similar problems. A model field gives a way to hide details of an object's concrete state. Figure 7 gives an example (taken from [7]) of the use of model fields: by updating the satisfies clauses, e.g., to `this.width = this.w` and `this.heigth = this.h`, `Rectangle` can be re-implemented with two `int`s `w` and `h`, without affecting the verification of other classes. For every model field **model** T f **satisfies** Q in a class C, a total function $\#f : C \rightarrow T$ (an *abstraction function*) is added to the specification language. In predicates in the specification, the expression $E.f$ is treated as syntactic sugar for $\#f(E)$. Abstraction function $\#f$ is axiomatized in the deduction system by an axiom $\forall \sigma \in \Sigma \bullet \; [\![\forall \texttt{this} : C \bullet Q]\!]\sigma$.[2] This axiom is not visible outside of C's module. The axiomatization problems we have described for method functions apply to abstraction functions as well: for the purpose of this paper, a model field f that satisfies predicate Q can be treated as a parameterless pure method with postcondition Q, with **result** for $\texttt{this}.f$.

Contributions. The contributions of this paper are the following:

1. We formalize and strengthen an implicit claim from [3]: No inconsistency is introduced by axioms of the form (2) $\Rightarrow$ (1) if every method function call in a pure method m's specification lies below m in a partial order $\prec$ (Sect. 2).

[2] More axioms might be added depending on the methodology, see Sect. 5.

2. We present a much improved scheme that leverages the power of the theorem prover to prove (2) once-and-for-all (Sect. 3).
3. We introduce a permissive definition for $\prec$ that improves the one in [3] and allows a greater degree of (mutual) recursion than before (Sect. 4).

We report on our experience and discuss related work in Sect. 5.

2 Avoiding Inconsistency

In this section, we identify proof obligations that allow axioms of form (1) to be added to the deduction system without introducing inconsistencies.

Let there be $N+1$ pure methods in the program fragment that is to be verified, labeled $m_0, \dots m_N$. For simplicity, assume that there are no static pure methods and that every pure method m_i has exactly one formal parameter $\mathtt{p}_i$ of type T'_i (extending to an arbitrary number of parameters is straightforward). Let T_i be the return type of pure method m_i. Let C_i be the class that defines m_i. Let predicates Pre_i and $Post_i$ be the pre- and postconditions of m_i. $PureAx$, defined below, represents the axioms introduced by pure methods (reformulated into a single proposition). We use $\equiv$ to define syntactical shorthands.

Definition 1 ($PureAx$).
$Spec_i \quad\equiv\quad Pre_i \Rightarrow Post_i$
$MSpec_i \quad\equiv\quad \forall \mathtt{this}\colon C_i,\ \mathtt{p}_i\colon T'_i \bullet Spec_i[\#m_i(\mathtt{this}, \mathtt{p}_i)/\mathbf{result}]$
$PureAx \quad\equiv\quad \forall \sigma \in \Sigma \bullet\ [\![MSpec_0 \wedge \dots \wedge MSpec_N]\!]\sigma$

Let $Prelude$ be the conjunction of all axioms in the deduction system that are not introduced by a pure method. The goal is to find proof obligations POs such that if $Prelude$ is consistent and POs hold, then adding the axioms for pure methods does not introduce inconsistencies. Theorem 1 formalizes this goal:

Theorem 1. $Prelude \Rightarrow (POs \Rightarrow PureAx)$

The remainder of this section discusses the proof obligations POs that we use to ensure that Thm. 1 holds. The theorem itself is proven to hold in the accompanying technical report [8]. If there is no recursion in pure method specifications, then Thm. 1 can be shown to hold using $POs \equiv PO1$ (see [4]):

Definition 2 ($PO1$).
$PO1_i \quad\equiv\quad \forall \sigma \in \Sigma \bullet\ [\![\forall \mathtt{this}\colon C_i,\ \mathtt{p}_i\colon T'_i \bullet \exists \mathbf{result}\colon T_i \bullet Spec_i]\!]\sigma$
$PO1 \quad\equiv\quad PO1_0 \wedge \dots \wedge PO1_N$

Note that $PO1_i$ is equivalent to proposition (2) from the introduction.

When there is (mutual) recursion, the crucial property that is in jeopardy is *functional consistency*: if the same function is called twice from the same state and the parameters of the two calls evaluate to the same values, then the two calls evaluate to the same value. For instance, consider the methods in Fig. 3. If pure methods add propositions of the form (1) to the deduction system, then these method definitions allow one to deduce that $\#\mathtt{n}(\mathtt{this}, \mathtt{i}) =$

$\#\mathtt{n}(\mathtt{this},\mathtt{i})+1$, which contradicts functional consistency of $\#\mathtt{n}$. More formally, since $[\![\#m_i(E_0,E_1)]\!]\sigma = \#m_i(\,[\![E_0]\!]\sigma,\ [\![E_1]\!]\sigma)$ (see Sect. 1), it follows immediately that $\forall \sigma \in \Sigma,\ i \in [0,N] \bullet [\![\forall c_0, c_1\colon C_i,\ p_0, p_1\colon T'_i \bullet c_0 = c_1 \wedge p_0 = p_1 \Rightarrow \#m_i(c_0,p_0) = \#m_i(c_1,p_1)]\!]\sigma$. The proof obligations must ensure that the axioms introduced by pure methods do not contradict functional consistency.

For convenience, we define the equivalence relation $\sim$:

Definition 3 ($\sim$).
$$[\![\#m_i(E_0,E_1) \sim \#m_j(E_2,E_3)]\!]\sigma \ \overset{def}{=}\ i = j \wedge\ [\![E_0 = E_2 \wedge E_1 = E_3]\!]\sigma$$

Then $\forall \sigma \in \Sigma \bullet [\![\#m_i(E_0,E_1) \sim \#m_j(E_2,E_3) \Rightarrow \#m_i(E_0,E_1) = \#m_j(E_2,E_3)]\!]\sigma$.

To ensure that recursive specifications do not lead to an axiomatization that contradicts functional consistency, we require the verifier to ensure that a function call in the axiomatization of $\#m_i(o,x)$ does not (indirectly) depend on the value of $\#m_i(o,x)$. To this end, we introduce the strict partial order $\prec$ on method function calls (i.e., $\prec$ is an irreflexive and transitive binary relation on expressions of the shape $\#m_i(E_0,E_1)$). The definition of $\prec$ is not relevant to the proof as long as (1) $\prec$ is well-founded, and (2) the following lemma holds:

Lemma 1. $\forall \sigma \in \Sigma,\ i,j \in [0,N]\ \bullet [\![\forall c_0\colon C_i,\ x_0\colon T'_i,\ c_1\colon C_j,\ x_1\colon T'_j \bullet$
$\#m_i(c_0,x_0) \prec \#m_j(c_1,x_1)\ \Rightarrow\ \#m_i(c_0,x_0) \not\sim \#m_j(c_1,x_1)]\!]\sigma$

In Sect. 4, we present a definition of $\prec$ that is suitable for our proof system. Proof obligation $PO2$, defined below, requires every method function call in the specification of m_i to lie below $\#m_i(\mathtt{this},\mathtt{p}_i)$ in the order $\prec$ in every state in which the result of the call is relevant.

Definition 4 ($PO2$). *Let $i,j \in [0,N]$. Let $NrOfCalls_{i,j}$ be the number of calls to $\#m_j$ in $Spec_i$. If $l+1 = NrOfCalls_{i,j}$, and $k \in [0,l]$, then*
$Call_{i,j,k}$ is the expression that is the k'th call to $\#m_j$ in $Spec_i$
$Spec_{i,j,k}$ is $Spec_i$, but with a fresh variable substituted for the k'th call to $\#m_j$

$$\begin{array}{lcl}
Smaller_{i,j,k} & \equiv & Call_{i,j,k} \prec \#m_i(\mathtt{this},\mathtt{p}_i) \\
NotRel_{i,j,k} & \equiv & \forall\, \mathbf{result}\colon T_i,\ x\colon T'_j \bullet Spec_{i,j,k} = Spec_i \\
PO2_{i,j,k} & \equiv & \forall \sigma \in \Sigma \bullet\ [\![\forall\, \mathtt{this}\colon C_i,\ \mathtt{p}_i\colon T'_i \bullet Smaller_{i,j,k} \vee NotRel_{i,j,k}]\!]\sigma \\
PO2_{i,j} & \equiv & PO2_{i,j,0} \wedge \ldots \wedge PO2_{i,j,l} \\
PO2_i & \equiv & PO2_{i,0} \wedge \ldots \wedge PO2_{i,N} \\
PO2 & \equiv & PO2_0 \wedge \ldots \wedge PO2_N
\end{array}$$

The intuition behind $NotRel$ is that $Call_{i,j,k}$ in $Spec_i$ is not relevant in $\sigma \in \Sigma$ if the result value of $Call_{i,j,k}$ is not relevant to the value of $\#m_i(\mathtt{this},\mathtt{p}_i)$ in σ. That is, for any value of **result**, the value of $Spec_i$ is the same for any result of $Call_{i,j,k}$. As an extreme example, suppose $Spec_i$ is `false` $\Rightarrow$ **result** `= this.`m_i`(p) + 1`. Then $Smaller_{i,i,0}$ never holds, but $NotRel_{i,i,0}$ always holds as $\forall \sigma \in \Sigma \bullet [\![\forall\, \mathtt{this}\colon C_i,\ \mathtt{p}_i\colon T'_i,\ \mathbf{result}\colon T_j,\ x\colon T'_j \bullet$ `(false` $\Rightarrow$ **result** `= this.`m_i`(p) + 1) = (false` $\Rightarrow$ **result** `= x + 1)` $]\!]\sigma$. Then $PO2_{i,i,0}$ is met, and hence $PO2_i$ is met. We show a more realistic example in Sect. 4.1.

In this section, we formalized the problem sketched in the introduction. Furthermore, we introduced high-level proof obligations that ensure that the extension of the *Prelude* with the axiomatization of pure methods does not introduce inconsistencies: in [8] we prove that Thm. 1 holds if $POs \equiv PO1 \wedge PO2$. In the next two sections, we address two remaining practical concerns: we provide heuristics to prove $PO1$, and define the partial ordering $\prec$ used in $PO2$.

3 Heuristics for Establishing $PO1$

Proof obligation $PO1$ poses serious difficulties for automatic verification. Even in simple cases, automatic theorem provers are unable to come up with a witness for the existential quantification $\exists\,\mathbf{result} : T_i \bullet \mathit{Spec}_i$ in $PO1_i$. As a solution, [3] proposes only to allow a pure method m_i when (1) it has a postcondition of the form **result** *op* E or E *op* **result**, where *op* is a binary operator from the set $\{=, \geq, \leq, \Rightarrow, \Leftrightarrow\}$, and (2) E is an expression that does not contain **result**. If these conditions are met, then E is a witness for the quantification, i.e., $\forall \sigma \in \Sigma \bullet\ [\![\forall \mathtt{this} : C_i, \mathtt{p}_i : T_i' \bullet \mathit{Spec}_i[E/\mathbf{result}]]\!]\sigma$, and therefore $PO1_i$ holds.

This solution has the advantage that it only requires a simple syntactic check. However, it is quite restrictive. Unfortunately, not much more can be done with syntactic checks. For instance, consider method `findInsertPosition` from Fig. 5. Here, 0 is a witness (as $\mathtt{0} \leq \mathtt{N} \Rightarrow \mathtt{0} \leq \mathtt{0} \wedge \mathtt{0} \leq \mathtt{N}$). However, a syntactic check cannot establish that $0 \leq \mathtt{N}$. Our solution is to leverage the power of the theorem prover. Consider the scheme below.

1. Find a witness candidate E.
2. If $\forall \sigma \in \Sigma \bullet\ [\![\forall \mathtt{this} : C_i, \mathtt{p}_i : T_i' \bullet \mathit{Spec}_i[E/\mathbf{result}]]\!]\sigma$ can be established by the theorem prover, then $PO1_i$ holds. Otherwise, the program is rejected.

This scheme is more powerful than the syntactic check of [3]. For instance, it allows `findInsertPosition`, assuming that 0 is found as a witness candidate. Before we discuss how to find witness candidates, we improve on the scheme above in one important way. Consider method `max` from Fig. 5. $PO1$ cannot be established for `max` using the scheme above, no matter which witness candidate is found. In particular, neither $\mathit{Spec}_{\mathtt{max}}[\mathtt{x}/\mathbf{result}]$ nor $\mathit{Spec}_{\mathtt{max}}[\mathtt{y}/\mathbf{result}]$ holds. The problem is that the scheme requires that there is a witness that holds in all cases. $PO1$ only requires that in all cases, there is a witness. The latter is true for `max`, but the former is not. If $\mathtt{x} \leq \mathtt{y}$, then `y` is a witness. If $\mathtt{y} \leq \mathtt{x}$, then `x` is a witness. That is, $\mathit{Spec}_{\mathtt{max}}[\mathtt{x}/\mathbf{result}] \vee \mathit{Spec}_{\mathtt{max}}[\mathtt{y}/\mathbf{result}]$ holds. Therefore, $\exists \mathbf{result} : \mathbf{int} \bullet \mathit{Spec}_{\mathtt{max}}$ holds, and $PO1$ holds. Based on this reasoning, the scheme presented above is replaced by the more liberal scheme below.

1. Find witness candidates $E_0, \ldots, E_n$.
2. If $\forall \sigma \in \Sigma \bullet\ [\![\forall \mathtt{this} \colon C_i,\ \mathtt{p}_i \colon T_i' \bullet \mathit{Spec}_i[E_0/\mathbf{result}] \vee \ldots \vee \mathit{Spec}_i[E_n/\mathbf{result}]]\!]\sigma$ can be established by the theorem prover, then $PO1_i$ holds. Otherwise, the program is rejected.

Next, we present an algorithm to find witness candidates for a pure method. We assume that there is a function $\mathit{kind} : \mathit{Type} \to \{\mathit{Bool}, \mathit{Enum}, \mathit{Num}, \mathit{Ref}\}$ that distinguishes four kinds of types. The algorithm uses a Haskell-like switch that uses pattern matching and does not fall through. For example, case A of B $\to$ C

D → E _ → F should be read as 'if A matches B, then C, else if A matches D, then E, else F'. The witness candidates for a pure method m_i with return type T_i and postcondition $Post_i$ are given by $wcs(T_i, Post_i)$. Below, wcs and its helper functions are defined, discussed and illustrated by a number of examples. Note that $ExprSet \equiv Set\ of\ Expression$, and that $|S|$ returns the size of set S.

Definition 5 (*wcs*)**.**

$wcs : Type \times Predicate \rightarrow ExprSet$

$wcs(T,P) \overset{def}{=}$ *case* $kind(T)$ *of*

$Bool \quad \rightarrow \{true, false\}$

$Enum \rightarrow$ *the enumerator list (i.e. the sequence of enumeration constants) of* T

$Ref \quad \rightarrow$ *let* $euld(P) = \langle S_0, S_1, S_2, S_3 \rangle$ *in* $S_0 \cup \{\texttt{null}\}$

$Num \quad \rightarrow$ *let* $euld(P) = \langle S_0, S_1, S_2, S_3 \rangle$ *in*

$\qquad S_0 \cup dupl(S_1, |S_3|, true) \cup dupl(S_2, |S_3|, false) \cup dupl(\{1\}, |S_3|, true)$

$euld : Predicate \rightarrow ExprSet \times ExprSet \times ExprSet \times ExprSet$

$euld(P) \overset{def}{=}$ *case* P *of*

$\mathbf{result} = E$ *or* $E = \mathbf{result} \rightarrow \langle \{E\}, \{\}, \{\}, \{\} \rangle$

$\mathbf{result} \geq E$ *or* $E \leq \mathbf{result} \rightarrow \langle \{\}, \{E\}, \{\}, \{\} \rangle$

$\mathbf{result} \leq E$ *or* $E \geq \mathbf{result} \rightarrow \langle \{\}, \{\}, \{E\}, \{\} \rangle$

$\mathbf{result} \neq E$ *or* $E \neq \mathbf{result} \rightarrow \langle \{\}, \{\}, \{\}, \{E\} \rangle$

$\mathbf{result} > E$ *or* $E < \mathbf{result} \rightarrow euld(\mathbf{result} \geq E + 1)$

$\mathbf{result} < E$ *or* $E > \mathbf{result} \rightarrow euld(\mathbf{result} \leq E - 1)$

$P_0 \vee P_1$ *or* $P_0 \wedge P_1 \quad \rightarrow$ *let* $\ euld(P_0) = \langle S_0, S_1, S_2, S_3 \rangle$

$\qquad\qquad$ *and* $\ euld(P_1) = \langle S'_0, S'_1, S'_2, S'_3 \rangle$ *in*

$\qquad\qquad\qquad \langle S_0 \cup S'_0, S_1 \cup S'_1, S_2 \cup S'_2, S_3 \cup S'_3 \rangle$

$\neg P_0 \qquad\qquad\qquad \rightarrow$ *let* $euld(P_0) = \langle S_0, S_1, S_2, S_3 \rangle$ *in*

$\qquad\qquad\qquad \langle S_3, addOrSub1(S2, true), addOrSub1(S1, false), S_0 \rangle$

$P_0 \Rightarrow P_1$ *or* $P_1 \Leftarrow P_0 \rightarrow euld(\neg P_0 \vee P_1)$

$P_0 \Leftrightarrow P_1 \qquad\qquad \rightarrow euld((P_0 \wedge P_1) \vee (\neg P_0 \vee \neg P_1))$

$P_0\ ?\ P_1 : P_2 \qquad\quad \rightarrow euld((P_0 \Rightarrow P_1) \wedge (\neg P_0 \Rightarrow P_2))$

$\_ \qquad\qquad\qquad\quad \rightarrow \langle \{\}, \{\}, \{\}, \{\} \rangle$

$addOrSub1 : ExprSet \times Bool \rightarrow ExprSet$

$addOrSub1(\{E_0, \ldots, E_n\}, isAdd) \overset{def}{=}$

$\quad (isAdd\ ?\ \{E_0 + 1, \ldots, E_n + 1\} : \{E_0 - 1, \ldots, E_n - 1\})$

$dupl : ExprSet \times \mathbb{N} \times Bool \rightarrow ExprSet$

$dupl(\{E_1, \ldots, E_n\}, duplCnt, isAdd) \overset{def}{=}$

$\quad duplExpr(E_1, duplCnt, isAdd) \cup \ldots \cup duplExpr(E_n, duplCnt, isAdd)$

$duplExpr : Expression \times \mathbb{N} \times Bool \rightarrow ExprSet$

$duplExpr(E, duplCnt, isAdd) \overset{def}{=}$

$\quad (isAdd\ ?\ \{E + 0, \ldots, E + duplCnt\} : \{E - 0, \ldots, E - duplCnt\})$

The intuition behind the $wcs(T,P)$ definition is as follows. If *kind(T)* $\in$ {*Bool, Enum*}, then there is no need to scan the postcondition for witness candidates. Instead, we make full use of the possibility to select multiple candidates and let every value of the type be a witness candidate. If *kind(T)* $\in$ $\{$*Num, Ref*$\}$, then function *euld* is used to scan P for *e*qualities, *u*pper bounds, *l*ower bounds, and *d*isequalities that contain **result**. More precisely, assume $\mathit{euld}(P) = (S_0, S_1, S_2, S_3)$. Let $\mathit{cnf}(P)$ yield the conjunctive normal form of P, and let $\mathit{test}(P, E) \equiv \mathit{cnf}(P)[E/\mathbf{result}]$. Let E_0, E_1, E_2 and E_3 be elements of S_0, S_1, S_2 and S_3, respectively. Then for every $n \in \mathbb{N}$, each of $\mathit{test}(P, E_0)$, $\mathit{test}(P, E_1 + n)$ and $\mathit{test}(P, E_2 - n)$ has a satisfied conjunct. Also, at least one conjunct of $\mathit{test}(P, E_3)$ contains an unsatisfied disjunct. From *euld*'s result, witness candidates are extracted and where needed duplicated using function *dupl*.

We illustrate with several examples. Let $\mathit{kind}(T_i) = \mathit{Num}$. If Post_i is **result** $= 4$, or **result** > 3, or **result** ≤ 4, then $\mathit{euld}(\mathit{Post}_i)$ is $\langle\{4\}, \{\}, \{\}, \{\}\rangle$, $\langle\{\}, \{4\}, \{\}, \{\}\rangle$, or $\langle\{\}, \{\}, \{4\}, \{\}\rangle$, respectively. In each case, $\mathit{wcs}(T_i, \mathit{Post}_i) = \{4, 1\}$. As $\mathit{Post}_i[4/\mathbf{result}]$ holds, $\mathit{Post}_i[4/\mathbf{result}] \vee \mathit{Post}_i[1/\mathbf{result}]$ holds as well and $PO1_i$ is satisfied. Default witness 1 is included to handle, e.g., the case where Post_i is **result** $\neq 4$. Then $\mathit{euld}(\mathit{Post}_i) = \langle\{\}, \{\}, \{\}, \{4\}\rangle$. Then $\mathit{wcs}(T_i, \mathit{Post}_i) = \{1, 2\}$. As $\mathit{Post}_i[1/\mathbf{result}]$ holds, $PO1_i$ is satisfied.

We track upper and lower bounds and the number of disequalities N to handle, e.g., the case where Post_i is $\mathbf{result} > 4 \wedge \mathbf{result} \neq 5$. Then $\mathit{euld}(\mathit{Post}_i) = \langle\{\}, \{5\}, \{\}, \{5\}\rangle$, and $\mathit{wcs}(T_i, \mathit{Post}_i) = \{5, 6, 1, 2\}$. As $\mathit{Post}_i[6/\mathbf{result}]$ holds, $PO1_i$ is satisfied. More generally, by trying N different candidates that all satisfy the bound, we are sure to find at least one that satisfies the disequality.

We combine the candidates found in subpredicates of conjunctions and disjunctions to handle, e.g., the case where Post_i is (**result** = 4 $\vee$ **result** $>$ 8) $\wedge$ **result** $>$ 7. Then $\mathit{euld}(\mathit{Post}_i) = \langle\{4\}, \{9, 8\}, \{\}, \{\}\rangle$, and $\mathit{wcs}(T_i, \mathit{Post}_i) = \{4, 9, 8, 1\}$. As $\mathit{Post}_i[9/\mathbf{result}]$ holds, $PO1_i$ is satisfied.

A predicate $\neg P$ is dealt with 'on the fly', which is more efficient than distributing the negation over the subexpressions of P. We interchange S_0 and S_3 as well as S_1 and S_2, and then add (subtract) 1 to each element of the new S_1 (S_2). The intuition is the following. As was stated above, if $E \in S_1$, then for every n a conjunct in $\mathit{test}(P, E + n)$ holds. Then for every n, a conjunct in $\mathit{test}(\neg P, E - 1 - n)$ holds. For example, $\neg(\mathbf{result} \geq E)$ equals $\mathbf{result} \leq E - 1$.

As an aside, note that in the cases where P is either $P_0 \Leftrightarrow P_1$ or P_0 ? P_1 : P_2, $\mathit{euld}(P_0)$ and $\mathit{euld}(P_1)$ are evaluated twice. These cases can be optimized at the expense of a more complicated definition.

4 Defining the Ordering $\prec$

Our definition of $\prec$ builds on work in [3, 4]. It uses a function *Order* (defined below) that associates a tuple of numbers with an expression $\#m_i(E_0, E_1)$ in a state σ. Our definition of $\prec$ ensures that $\prec$ is a well-founded strict partial order, and that Lem. 1 holds (as long as *Order* is well-defined):

Definition 6 ($\prec$). $[\![\#m_i(E_0, E_1) \prec \#m_j(E_2, E_3)]\!]\sigma \overset{def}{=}$
$\mathit{Order}(\#m_i, E_0, E_1, \sigma)$ is lexicographically ordered below $\mathit{Order}(\#m_j, E_2, E_3, \sigma)$

As Def. 7 shows, the definition of *Order* uses three functions. *RootDistance* associates a number with an object based on the well-founded strict partial order on objects provided by ownership, an existing specification technique (Sect. 4.1). *RTVal* associates a number with a method function based on a numbering scheme that can be largely inferred automatically (Sect. 4.2). *MeasuredBy* yields a tuple of numbers that is determined by a pure method's `measuredBy` clause, and that depends only on the values of the numerical parameters (Sect. 4.3). The definition uses $\triangleright$ to denote sequence concatenation.

Definition 7 (*Order* **).**
$\mathit{Order} : \mathit{Method\ Function} \times \mathit{Expression} \times \mathit{Expression} \times \Sigma \rightarrow \mathit{Sequence\ of\ } \mathbb{Z}$
$\mathit{Order}(\#m_i, E_0, E_1, \sigma) \overset{def}{=} \langle j, \mathit{RTVal}(\#m_i)\rangle \triangleright \mathit{MeasuredBy}(\#m_i, E_1, \sigma)$,
where j *is* ($[\![E_0]\!]\sigma \in \texttt{Object}$? $-\mathit{RootDistance}(\,[\![E_0]\!]\sigma, \sigma) : 0)$.

Note that if $\#m_j(E_0, E_1)$ occurs in the specification of m_i, and σ does not map E_0 to an object, then the first element of $\mathit{Order}(\#m_j, E_0, E_1, \sigma)$ is 0, thus requiring that the call is not relevant in σ if $PO2$ is to hold.

4.1 Root Distance

Ownership, originally developed to enforce state encapsulation [9, 10], is a commonly used technique to make whole/part relations explicit in specifications (often applied to the modular verification of invariants [11–14]). The set of *owners* consists of the set of objects and the special purpose owner **root**. In any given state, every object x is *directly owned* by exactly one owner o, $o \neq x$. The *owned* relation is the transitive closure of the directly owned relation. The intention is that an object x owns the objects that are part of x, i.e., that belong to x's representation. Objects that are not part of any other object are directly owned by **root**. The owned relation is required to be irreflexive, as a whole is not a part of one of its parts. Therefore, ownership is a well-founded strict partial order, which makes it suitable for use in the definition of $\prec$.

In [4], it is suggested that 'the height of an object in the ownership hierarchy' can be used to allow direct recursion. We formalize this notion and apply it to general recursion. The owned relation ensures that every object is owned by **root**. Let function $\mathit{RootDistance} : \mathit{Object} \times \Sigma \rightarrow \mathbb{N}$ be such that $\mathit{RootDistance}(x, \sigma) = n$ iff x is owned by exactly n objects in σ (we say x has *RootDistance* n in σ). Then *RootDistance* induces a well-founded strict partial order that is an extension of ownership: if object x is owned by object y in state σ, then $\mathit{RootDistance}(x, \sigma) > \mathit{RootDistance}(y, \sigma)$. Additionally, *RootDistance* orders objects that are not ordered by ownership. For instance, if x and y have the same direct owner in state σ, and object z is owned by y, then x and z are not ordered by ownership, but $\mathit{RootDistance}(x, \sigma) < \mathit{RootDistance}(z, \sigma)$.

Note that given Definitions 6 and 7, $[\![\#m_i(E_0, E_1) \prec \#m_j(E_2, E_3)]\!]\sigma = \mathit{true}$ when $[\![E_0]\!]\sigma = x$, $[\![E_2]\!]\sigma = y$, and $\mathit{RootDistance}(x, \sigma) > \mathit{RootDistance}(y, \sigma)$.

It is not necessary, and usually not possible, to determine an object's absolute *RootDistance* during static program verification. Rather, if m_i's specification

```
class Holding {
 rep Node myComps;
 rep Personnel myPnel;
} //rest of class omitted

class Company {
 peer Personnel thePnel;

 pure Node myPersonnel()
  ensures ∀ Person p • (
   result.has(p) ⇔
   this.thePnel.myPers.has(p) ∧
   p.worksFor ≠ null ∧
   p.worksFor.has(this) );
} //rest of class omitted
```

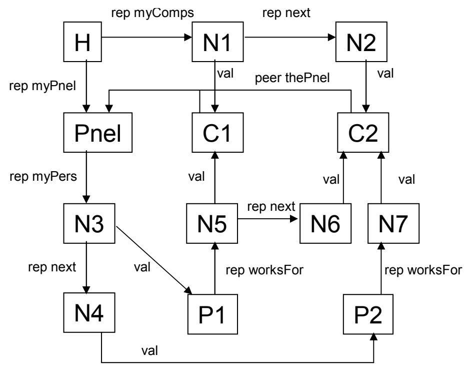

Fig. 8. Administration System. H is a `Holding`, Pnel a `Personnel`, N's are `Nodes`, C's `Companies`, and P's `Persons`. Person P2 works only for C2, and P1 works for both C's.

contains a call $\#m_j(E_0, E_1)$, one has to establish that the *RootDistance* of the `this`-object is smaller than (or at least equal to) the *RootDistance* of the E_0-object. I.e., one reasons about the relative *RootDistance*. This involves reasoning about ownership, which is often made explicit by extending types with *ownership modifiers* [15] like **rep** and **peer**. Consider a state σ in which an object x has a field f that refers to an object y. If the ownership modifier of f is **rep**, then x directly owns y and $\mathit{RootDistance}(y,\sigma) = \mathit{RootDistance}(x,\sigma)+1$. If it is **peer**, then x and y have the same direct owner and $\mathit{RootDistance}(y,\sigma) = \mathit{RootDistance}(x,\sigma)$. Alternatively, ownership can be encoded into existing proof system concepts using a specification-only field `owner` [12]. If x`.owner` evaluates to y in σ, then $\mathit{RootDistance}(x,\sigma) = \mathit{RootDistance}(y,\sigma)+1$.

The use of *RootDistance* is illustrated by method `Node.count` in Fig. 4. Its specification contains one call, to $\#\mathtt{count}(\mathtt{this.next}, \mathtt{obj})$. There are two cases, each of which satisfies $PO2$: if $[\![\mathtt{this.next} = \mathtt{null}]\!]\sigma = \mathit{false}$, modifier **rep** on `next` allows the verifier to deduce that $\mathit{RootDistance}(\,[\![\mathtt{this.next}]\!]\sigma,\sigma) = \mathit{RootDistance}(\,[\![\mathtt{this}]\!]\sigma,\sigma)+1$; if $[\![\mathtt{this.next} = \mathtt{null}]\!]\sigma = \mathit{true}$, *NotRel* holds as $[\![(\mathtt{this.next} = \mathtt{null}\ ?\ 0 : \mathtt{this.next.count(obj)})]\!]\sigma = 0$, which means that the value of $[\![\#\mathtt{count}(\mathtt{this.next}, \mathtt{obj})]\!]\sigma$ is not relevant.

The extension of ownership provided by *RootDistance* is useful for non-hierarchical scenarios. For instance, Fig. 8 shows two classes and a possible object configuration of an administration system. In this system, a `Holding` consists of multiple `Companies`, and a `Person` that is part of the `Holding` can work for multiple of these `Companies`. A `Personnel` object manages (access to) these `Persons`. Classes `Personnel` and `Person` are omitted. Each has only one relevant field. `Personnel` has a field **rep** `Node myPers` which refers to a linked list of the `Persons`. `Person` has a field **rep** `Node worksFor` which refers to a linked list of the `Companies` that `Person` works for. Class `Node` is found in Fig. 4. Pure method `Company.myPersonnel` returns a linked list of all `Persons` that work for that

`Company` (e.g., if called on C1, it returns a single node with `val` P1). Assume that it can be deduced that `Company.thePnel` and `Personnel.thePers` are never `null` (for instance because of an invariant or non-null annotation [16]). Then the `this.thePnel.myPers.has(p)` call in `Company.myPersonnel` is allowed as `myPers` is a rep field of a peer of `this` and thus has a higher *RootDistance*. More formally, in any state $\sigma \in \Sigma$ in which `this` evaluates to a `Company` object, $\mathit{RootDistance}(\ [\![\texttt{this.thePnel.myPers}]\!]\sigma, \sigma) = \mathit{RootDistance}(\ [\![\texttt{this}]\!]\sigma, \sigma) + 1$, and therefore, $[\![\#\texttt{has}(\texttt{this.thePnel.myPersons}, \texttt{p}) \prec \#\texttt{myPersonnel}(\texttt{this})]\!]\sigma$ holds. Likewise, the `p.worksFor.has(p)` call is allowed if one can deduce that the `Persons` in the list maintained by `p.thePnel` are owned by `p.thePnel` or by the `Holding` that owns `p.thePnel`. We discuss the **result**`.has(p)` call in Sect. 5.

4.2 Recursion Termination Value

For the second ordering, a *Recursion Termination Value* (RTV) is associated with each pure method [3]. A RTV is an element of the interval $[0, \mathit{maxRTV}]$, where maxRTV is a sufficiently large constant, e.g. maxInt. $\mathit{RTVal}(\#m_i)$ yields the RTV associated with pure method m_i.

Note that given Definitions 6 and 7, $[\![\#m_i(E_0, E_1) \prec \#m_j(E_2, E_3)]\!]\sigma = \mathit{true}$ when $\mathit{RootDistance}(\ [\![E_0]\!]\sigma, \sigma) = \mathit{RootDistance}(\ [\![E_2]\!]\sigma, \sigma)$ and $\mathit{RTVal}(\#m_i) < \mathit{RTVal}(\#m_j)$.

The RTV can be specified explicitly. For instance, in Spec# the RTV is specified by the `RecursionTermination` attribute that takes an integer parameter. The main advantage of the RTV ordering, however, is that it is largely inferred automatically. This inference is complicated by the desire for modular development (see Sect. 1).

Of course, the goal of the inference is to assign a RTV to every $\#m_i$ such that for every i, the inferred RTV is high enough to conclude $PO2_i$. When the the specification of m_i is changed, the previously inferred RTV for $\#m_i$ might no longer be high enough (for instance, because the specification of m_i now contains a method call). Therefore, the inference is rerun prior to re-verification. But as a consequence of modular development, it is not possible to re-infer every RTV. In particular, a RTV in a module that is hidden cannot be re-inferred. As a consequence, if an inferred RTV were publicly visible, a change to a specification that is hidden from a module M could indirectly invalidate the verification of M. That is, suppose that m_i and m_j are defined in different modules, and that the proof of $PO2_i$ depends on $\mathit{RTVal}(\#m_j) = n$. Suppose a part of the specification of m_j that is hidden from m_i is changed in such a way that re-inference of $\mathit{RTVal}(\#m_j)$ changes it to $n + 1$. Then the proof of $PO2_i$ no longer holds. While this does not go against modular development technically (re-inference of $\mathit{RTVal}(\#m_j)$ constitutes a change of public part of the specification of m_i), it is not intuitive (as the change is to an *implicit* part of the specification). Therefore, an inferred RTV is private, and an explicitly specified RTV is public. As the specifier has committed to the RTV, it is intuitive that changing it will require re-verification of modules to which it is visible. We discuss an algorithm to infer the RTVs for a module M in [8]. The outline is as follows. Construct a directed graph with a node N for every method visible in M, and with an edge from N to node N' iff N' occurs in the specification of N. For every N with an explicitly specified $RTV\ i$, label N with i. For every N with an RTV that is hidden from M, label N with $maxRTV$. For every remaining N, label N with the lowest value such that (1) N cannot reach a node with a higher RTV, and (2) if possible, such that N cannot reach a node with the same RTV. (1) is always possible, as $maxRTV$ can be assigned to all nodes. (2) can't be achieved for nodes that are part of a cycle, nor for nodes that can reach a $maxRTV$ node.

4.3 The measuredBy Clause

The third ordering allows for directly or mutually recursive method functions. We associate with pure method m_i, a *measuredBy clause* that specifies a tuple of numerical expressions $\langle E_1, \ldots, E_n \rangle$. $MeasuredBy(\#m_i, E, \sigma)$ is defined as $\langle\ [\![E_1[E/\mathtt{p}_i]]\!]\sigma, \ldots,\ [\![E_n[E/\mathtt{p}_i]]\!]\sigma \rangle$. For each such expression E_j, there is a proof obligation that $Pre_i \Rightarrow 0 \leq E_j$, which ensures that the ordering is well-founded. We restrict the free variables in these expressions to be the numerical formal parameters of m_i, but one can easily imagine allowing other variables, too, for example so that one can mention the *RootDistance* of a non-`this` object parameter. By default, the `measuredBy` clause is tuple $\langle 0 \rangle$.

The use of the `measuredBy` clause is illustrated by Fig. 6, where it allows the mutually recursive methods `isEven` and `isOdd`. For the call to `this.isOdd(n-1)` in the specification of `isEven`, the reasoning is as follows. Consider an arbitrary $\sigma \in \mathit{\Sigma}$. Assume $r_0, r_1, r_2, t_0, t_1, t_2 \in \mathbb{Z}$ such that $Order(\#isEven, \mathtt{this}, \mathtt{n}, \sigma) = \langle r_0, r_1, r_2 \rangle$, and $Order(\#isOdd, \mathtt{this}, \mathtt{n}-1, \sigma) = \langle t_0, t_1, t_2 \rangle$. Then $r_0 = t_0$, as both are determined by the *RootDistance* of the `this`-object (see Sect. 4.1). Also, $r_1 = t_1$ as the same RTV is assigned to mutually recursive method functions (see Sect. 4.2). Finally, $r_2 > t_2$ as $r_2 = [\![2n]\!]\sigma$, and $t_2 = [\![(2m+1)[n-1/m]]\!]\sigma = [\![2n-1]\!]\sigma$. Thus, $\langle t_0, t_1, t_2 \rangle$ is ordered lexicographically below $\langle r_0, r_1, r_2 \rangle$. So, if `C` is the class that declares `isEven` and `isOdd`, then $\forall \sigma \in \mathit{\Sigma} \bullet [\![\forall\, \mathtt{this} : \mathtt{C}, \mathtt{n} : \mathtt{int} \bullet \#isOdd(\mathtt{this}, \mathtt{n}-1) \prec \#isEven(\mathtt{this}, \mathtt{n})]\!]\sigma$. For the call to `isEven(m)` in the specification of `isOdd`, the reasoning is similar (the essential observation being that $2m+1 > (2n)[m/n]$). Together, these properties establish that $PO2$ holds.

5 Related Work and Experience

Frame properties for a model field f declared in a class C (see Sect. 1) are discussed in [7]. Essentially, the idea is to add a specification-only field f to C, and to extend the deduction system with a second axiom $\forall \sigma \in \mathit{\Sigma} \bullet [\![\forall \mathtt{this}\colon C \bullet P \Rightarrow \mathtt{this}.f = \#f(\mathtt{this})]\!]\sigma$, where P (defined by the methodology) describes the conditions under which the relation should hold. The methodology ensures that that $\#f(\mathtt{this})$ is assigned to f whenever P becomes *true*. Breunesse and Poll suggest desugaring a model field using its satisfies clause [17]. This simplifies the treatment of model fields considerably, but does not account for recursion or for visibility constraints on satisfies clauses.

Modeling partial functions by underspecified total functions in the underlying logic can lead to unintuitive outcomes for the users of the specification language [18]. Recent work by Rudich *et al.* [19] discusses how to prevent such outcomes. The work also discusses how to allow conditional use of the axioms introduced by pure methods, as well as class invariants, when establishing $PO1$ (see Sect. 2). Essentially, the idea is that if $Smaller_{i,j,k}$ holds, then the axiom introduced by m_j, instantiated for $Call_{i,j,k}$, can be assumed when proving $PO1_i$ (see Definitions 2 and 4). More formally, let $P_{i,j,k} \equiv (Smaller_{i,j,k}?Spec_j[Call_{i,j,k}/\mathbf{result}] : true)$. Then $PO1_i$ can be weakened to $\forall \sigma \in \Sigma \bullet [\![\forall \mathtt{this}\colon C_i,\ \mathtt{p}_i\colon T'_i \bullet P \Rightarrow \exists \mathbf{result}\colon T_i \bullet Spec_i]\!]\sigma$, where P consists of a conjunct $P_{i,j,k}$ for every $i, j \in [0, N]$, for every $k \in [0, NrOfCalls_{i,j} - 1]$.

In Sect. 4.1, we discussed how our approach allows a number of the calls in the specification of the `myPersonnel` method in Fig. 8. The call to **result**`.has()` in that specification, however, is problematic. The axiom introduced by the pure method describes a property that holds in every well-formed program state. Therefore, the resulting list of `Node`s has to exist in each such state (and contain the right `Person`s). This is reflected in $PO1$, which cannot be proven to hold for this example. Possible solutions to this problem are suggested in [4, 20, 21].

The heuristic guesses of candidate witnesses and the accompanying semantics checks in this paper have been implemented in the Spec# programming system; there is a partial implementation of RootDistance and the RTV scheme [3]. Pure methods occur frequently in practice, partly because Spec# by default treats property getters as pure methods. The Spec#/Boogie test suite alone requires 148 consistency checks. From more than a year's use, we find that, with one exception, the heuristics adequately guess candidate witnesses that (for consistently specified pure methods) the semantic checks quickly verify to ensure consistency.

The one exception to this positive experience has been pure methods with a non-null return type. The only non-null candidate witnesses that our heuristics guess are fields or parameters of exactly those types—the heuristics cannot use calls to constructors, as this would require one to first prove the consistency of the specifications of such constructors. Luckily, this case has occurred only for property getters whose body returns a newly allocated object (see [22] for a technique that allows such methods to be considered observationally pure). In the cases we have found, these property getters were not used as pure methods, so we could circumvent the problem by explicitly marking them non-pure.

6 Conclusions

Pure methods and model fields are useful and common specification constructs that can be interpreted by the introduction of axioms in the underlying proof system. Care has to be taken that these axioms do not introduce an inconsistency into the proof system. In this paper, we described and proved sound an approach that ensures no inconsistencies are introduced, and we described heuristics for the part of the approach that is problematic for trigger-based SMT solvers.

References

1. Cok, D.R.: Reasoning with specifications containing method calls and model fields. Journal of Object Technology **4** (2005) 77–103 FTfJP'04 Special Issue.
2. Cheon, Y., Leavens, G., Sitaraman, M., Edwards, S.: Model variables: cleanly supporting abstraction in design by contract. Softw. Pract. Exper. **35** (2005) 583–599
3. Darvas, Á., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: FASE. Volume 4422 of LNCS., Springer (2007) 336–351
4. Darvas, Á., Müller, P.: Reasoning About Method Calls in Interface Specifications. Journal of Object Technology **5** (2006) 59–85 FTfJP'05 Special Issue.
5. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52** (2005) 365–473
6. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. Volume 4963 of LNCS., Springer (2008) 337–340
7. Leino, K.R.M., Müller, P.: A verification methodology for model fields. In: ESOP. Volume 3924 of LNCS., Springer (2006) 115–130
8. Leino, K.R.M., Middelkoop, R.: Proving consistency of pure methods and model fields. Technical report, Microsoft Research (2009)
9. Clarke, D.: Object Ownership and Containment. PhD thesis, University of New South Wales (2001)
10. Müller, P.: Modular Specification and Verification of Object Oriented Programs. Volume 2262 of LNCS. Springer (2002)
11. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology **3** (2004) 27–56 FTfJP'03 Special Issue.
12. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: ECOOP. Volume 3086 of LNCS., Springer (2004) 491–516
13. Middelkoop, R., Huizing, C., Kuiper, R., Luit, E.J.: Specification and Verification of Invariants by Exploiting Layers in OO Designs. Fundamenta Informaticae **85** (2008) 377–398 CS&P'07 Special Issue.
14. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Science of Computer Programming **62** (2006) 253–286
15. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA '98, ACM Press (1998) 48–64
16. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: OOPSLA, ACM (2003) 302–312
17. Breunesse, C.B., Poll, E.: Verifying JML specifications with model fields. In: FTfJP'03, Technical Report 408, ETH Zurich (2003) 51–60
18. Chalin, P.: Are the logical foundations of verifying compiler prototypes matching user expectations? Form. Asp. Comput. **19** (2007) 139–158
19. Rudich, A., Darvas, Á., Müller, P.: Checking well-formedness of pure-method specifications. In: Formal Methods. Volume 5014 of LNCS., Springer (2008) 68–83
20. Naumann, D.A.: Observational purity and encapsulation. Theor. Comput. Sci. **376** (2007) 205–224
21. Barnett, M., Naumann, D.A., Schulte, W., Sun, Q.: 99.44% pure: Useful abstractions in specifications. In: FTfJP'04, Technical Report NIII-R0426, University of Nijmegen (2004) 11–18
22. Leino, K.R.M., Müller, P.: Verification of equivalent-results methods. In: ESOP. Volume 4960 of LNCS., Springer (2008) 307–321