Practical Reasoning About Invocations and Implementations of Pure Methods

Ádám Darvas[1] and K. Rustan M. Leino[2]

[1] ETH Zurich, Switzerland
adam.darvas@inf.ethz.ch
[2] Microsoft Research, Redmond, WA, USA
leino@microsoft.com

Abstract. User-defined functions used in the specification of object-oriented programs are called *pure methods*. Providing sound and practical support for pure methods in a verification system faces many challenges, especially when pure methods have executable implementations and can be invoked from code at run time. This paper describes a design for reasoning about pure methods in the context of sound, modular verification. The design addresses (1) how to axiomatize pure methods as mathematical functions enabling reasoning about their result values; (2) preconditions and frame conditions for pure methods enabling reasoning about the implementation of a pure method. Two important considerations of the design are that it work with object invariants and that its logical encoding be suitable for fully automatic theorem provers. The design has been implemented in the Spec# programming system.

1 Introduction

The notion of using user-defined functions in program specifications is both natural and useful. In object-oriented languages, such functions are known as *pure methods*—"pure" because their evaluation does not have an effect on the caller's program state [11]. The use of pure methods facilitates abstraction. To illustrate that, consider the postcondition

ensures this.IsPremium() ==> balance == **old**(balance) * 1.02;

of a method Bonus in an Account class. The condition expresses that if the Account object is a "premium" account, then its balance gets increased by 2%. The advantage of using pure method IsPremium in the condition is that its specification need not be repeated, which makes the condition more concise and comprehensible. Furthermore, even if the meaning (that is, specification and implementation) of the method changes over time, the postcondition of method Bonus need not be modified.

State-of-the-art languages such as Eiffel [15], Java with JML [11], and Spec# [3] support the use of pure methods. Common to these specification-enriched object-oriented languages is that their specifications are executable, and in particular their pure methods are executable. Like ordinary methods, pure methods

M.B. Dwyer and A. Lopes (Eds.): FASE 2007, LNCS 4422, pp. 336–351, 2007.

have associated pre- and postconditions that we use in static verification to reason about invocations and implementations of the methods. But pure methods are different from ordinary methods in three ways.

First, pure methods can be used in specifications, which are expressions to assert properties of the code in a program. Since code also contains expressions, it is tempting to treat specifications like other expressions appearing in the code. But if the evaluation of specifications were treated as ordinary code, then it would be difficult to allow quantifications in specifications: quantifications in code are treated as loops, to reason about loops one needs loop invariants, and expressing these loop invariants, in general, requires quantifiers! So, we would like to treat specifications like mathematical expressions, not like code with control flow and side effects. And this means that we want to treat pure-method invocations in specifications like mathematical functions, not like procedure calls with side effects. Hence, when reasoning about the program, we introduce a mathematical function for every pure method, and this entails the task of turning the pure method's pre- and postconditions into well-founded mathematical definitions.

Second, to reason about invocations of pure methods (in our experience, more so than for ordinary methods), it is useful to specify the *read effects* of pure methods. Such specifications restrict what the result value of a pure method may depend on, which lets callers determine if various state changes have any effect on the pure method.

Third, there are situations in code where it is appropriate to allow a pure method to be invoked, but not an ordinary method. Such invocations thus require that pure methods be given weaker (*i.e.*, more applicable) preconditions than ordinary methods. We have found that with these weaker conditions, the abstraction techniques used to formulate frame conditions for ordinary methods lose too much precision when reasoning about implementations of pure methods.

In this paper, we describe the design of pure methods in the static program verifier Boogie [2] for Spec# [3]. The design addresses the issues mentioned above. It also addresses an important engineering issue arising from the fact that Boogie uses a fully automatic theorem prover, Simplify [7]. This automation comes at a price; relevant to our work here is the incompleteness of the handling of quantifiers, and specifically the prover's apparent inability to deal with reachability and to find witnesses for existential quantifiers.

We call our design *practical* because it is simple to realize and is capable of handling all cases we encountered in practice. However, it is not *universal* in the sense that there are programs that our design cannot handle. Nonetheless, we believe that the design is a practically useful solution. Parts of the design that are not specific to the Boogie methodology can be adapted to work with other program verifiers too, for example, Krakatoa [14]. The parts that are specific to the Boogie methodology may not be directly adaptable to some other verifiers, but the issues that those parts of the paper bring out are still relevant to any verification system aiming for modular reasoning.

We have implemented our design in Boogie, which has allowed us to start experimenting with its practicality. One of the benchmarks we have used is the

source code of Boogie itself, which comprises some 45K lines of Spec#. Although much work remains to actually specify and verify the Boogie source code, all uses of pure methods in the source code pass our admissibility checks.

Section 2 briefly sketches the basic encoding and axiomatization of pure methods. Section 3 presents our design for the axiomatization which takes into account the constraints of Spec#. In Section 4, we give an overview of the Boogie methodology to prepare the reader for the remaining sections. Sections 5 and 6 study typical situations that occur during the verification of pure methods and propose axioms for their treatment. Finally, we list related work and conclude.

2 Encoding of Pure Methods and Their Return Values

In this section, we describe how methods in an object-oriented program are encoded as methods with first-order pre- and postconditions, uninterpreted functions, and a variety of axioms. These encodings can then be subjected to standard verification techniques based on, for example, Hoare logic [17] or verification-condition generation [14,2]. As is standard, our encoding represents the heap explicitly, so where the object-oriented notation would write $o.f$ to denote the f field of an object referenced by o, our encoding writes $h[o, f]$, which denotes the value at location (o, f) in the object store (that is, heap) h.

Pure methods and their uses in specifications are encoded by uninterpreted functions and function applications, respectively [5,6]. For each pure method M, the encoding introduces a function $\#M$, which we refer to as a *method function*. The method function takes one argument for each parameter of the method, including the receiver parameter, and an additional argument for the object store in which it is evaluated. For instance, a pure method declared as:

```
[Pure] int Exmpl(int x)
  requires this.f <= x;
  ensures result <= x - this.f;
{ return (x - this.f) / 2; }
```

where **result** denotes the return value of the method, introduces a method function $\#Exmpl$ with the signature $ref \times int \times heap \to int$, where ref, int, and $heap$ are the sorts for references, machine integers, and object stores, respectively. We can use a function, because it is standard to require pure methods to be deterministic [11].

Pure methods and their implementations also give rise to method declarations in the encoding. These method declarations are used to reason about calls to pure methods from code and to reason about the implementations of pure methods. To make a connection between calls to a pure method M in code and calls to M in specifications, we follow [5] and our encoding adds a postcondition that ties the result of M to the value of $\#M$. This is needed when the method specification does not completely determine the result value. For method Exmpl, the additional postcondition in the encoding is $\mathbf{result} = \#Exmpl(\mathbf{this}, x, h)$;

where h denotes the object store. This postcondition allows us to verify code like: "v = o.Exmpl(23); **assert** v == o.Exmpl(23);" in the source program, since the expression in the assert statement is a specification expression.

Method function $\#M$ is axiomatized based on the specification of method M. In essence, the axiom states that if the precondition of the method holds, then the method returns a value that is consistent with the postcondition. Formally, for a pure method M with pre- and postconditions *Pre* and *Post*, we would like an axiom like:

$$(\forall\, t, p, h \bullet \ Pre[t/this] \ \Rightarrow \ Post[t/this,\ \#M(t, p, h)/result]\ ) \tag{1}$$

where t and p range over the possible values of the receiver object and other parameters, respectively, and h ranges over well-formed heaps. The substitutions replace every occurrence of **this** by the bound variable for the receiver object and every occurrence of the special variable **result** by the function application of $\#M$. For method Exmpl, the axiom (after substitutions) is:

$$(\forall\, t, x, h \bullet \ h[t, f] \leqslant x \ \Rightarrow \ \#Exmpl(t, x, h) \leqslant x - h[t, f]\ )$$

Unfortunately, formula (1) does not necessarily give rise to a well-founded definition for $\#M$, because the postcondition *Post* might be unattainable (for example, *Post* might be *false*) [6]. This is not a problem for a verification system when verifying ordinary code, because it will not be possible to prove the correctness of any terminating path in the implementation. However, an unattainable postcondition would be a problem for a verification system if it caused the generation of an unsound axiom, because such an axiom would let every part of the program be verified, including the implementation of the pure method.

Formula (1) *is* sound if for all relevant values of the arguments, there is some value that $\#M$ can take on. More precisely, if M's specification does not call any pure methods, then the soundness of (1) follows from:

$$(\forall\, t, p, h \bullet \ Pre[t/this] \ \Rightarrow \ (\exists\, w \bullet \ Post[t/this,\ w/result]\ )) \tag{2}$$

Thus, in this case, a sound axiomatization of method function $\#M$ is the implication (2) $\Rightarrow$ (1). If M's specification calls pure methods, then soundness additionally requires that any recursion be well-founded, which can be expressed by a much more complicated form of (2) that takes into account the values of all applications of all method functions involved in the recursion.

3 Practical Issues of Method Functions

The idea of using (2) $\Rightarrow$ (1) as the axiom that defines the method function $\#M$ suffers from two practical problems, which we describe in this section. We also describe how we overcome these problems in our design, which is an adaptation of previous work by Darvas and Müller [6].

3.1 Well-Founded Definitions of Method Functions

One problem with using (2) $\Rightarrow$ (1) to axiomatize $\#M$ is the alternating quantifiers in (2), which do not work well with our automatic theorem prover. Some

simple experiments we conducted showed that Simplify does not discover even very simple witnesses w for the existential quantifier. In this section, we show how some simple syntactic checks guarantee the existence of a witness, thus satisfying the antecedent (2) once and for all.

As a first step in our heuristic process, we identify a *candidate witness expression* for the pure method. Then, we check that the candidate witness expression lies below the method function in a well-founded order (that is, a partial order with no infinite descending chains). If this process yields a well-founded candidate witness expression, then (2) is satisfied, so our encoding will simply include (1) as the axiom that constrains $\#M$. Since the heuristics look for syntactic patterns, they might not always discover a witness even if a witness does exist.

Identifying a candidate witness expression. Our heuristics consider a method to have a candidate witness expression E if and only if the method has exactly one **ensures** clause and it has the form "**result** *op* E" or "E *op* **result**", where *op* is one of the reflexive operators $==$, $\leqslant$, $\geqslant$, $\Rightarrow$, or $\Leftrightarrow$, and E is an expression that does not contain the literal **result**. The syntactic nature of the heuristics might require postconditions to be rewritten in order to have the witness discovered.

Expression ordering. An expression E lies below (an application of) a method function $\#M(t, p, h)$ if and only if every method function in E lies below $\#M(t, p, h)$. Method functions are ordered by a lexicographic ordering: method function $\#N(s, q, h)$ lies below $\#M(t, p, h)$ if and only if:

- s is the term $h[t, f]$ where f is a field declared with the **rep** modifier (we explain later what **rep** fields are; for now, it suffices to know that **rep** fields induce a well-founded order on the objects in any program state), or
- s and t are the same term, pure methods N and M are declared in the same class, and that class orders N before M.

A suitable well-founded ordering on pure methods within a class, if one exists, can be inferred by building a call graph based on how the postconditions of the pure methods of the class call each other [6]. Our design uses a cruder inference that associates a number with each method and orders the methods accordingly. We let programmers override this inference by marking a pure method with a new attribute called RecursionTermination that takes a natural number as parameter. The inference uses the programmer-specified number, if present; otherwise, if the method's postconditions does not call any other pure methods, the inferred number is 0; otherwise, the inferred number is infinity.

Our experience with the specification of Boogie suggests that the annotation overhead may not be too bad in practice. We had to annotate only one method in mscorlib, the Microsoft Common Object Runtime Library. This was partly due to the simple yet helpful inference mechanism. The advantage of the approach is its simplicity compared to building and analyzing the call graph of each class.

3.2 Tension Between Dynamic Execution and Static Verification

The other problem stems from a tension between our desire to encode a pure-method invocation in a specification as an application of the corresponding method function, which does not alter the heap, and our desire to allow the implementation of a pure method, which will be executed at run time, to make some changes to the heap. In this section, we describe that problem and the solution our design uses.

Pure methods are side-effect free. That is, they are not allowed to change the state of existing objects. However, in many practical cases it is convenient to permit them to allocate new objects and freely modify their state. For example, a pure method to find the median value in a collection may allocate a temporary *List* object and apply a number of mutating operations to it as part of computing the result of the pure method. This means that a call to a pure method may change the heap, but only in a way that does not change the state of objects that were allocated before the call.

We would like our encoding to evaluate each specification expression, including all the method calls it contains, in a single state. But this raises some concern, since at run time the pure methods may be invoked in different states. To illustrate that this is a real concern, consider the specification expression "o.M() == o.M()", where the implementation of pure method M allocates and returns a new object. At run time, the specification evaluates to *false*, but its encoding $\#M(o,h) = \#M(o,h)$ equals *true* [6].

One way to overcome this problem is to disallow pure methods to *return* newly allocated objects—while still permitting the allocation and modification of new objects [6]. This restriction makes the encoding indistinguishable from the run-time behavior, because the objects allocated in a call are not observable by specification expressions evaluated after the call. Although this solution leads to a simple encoding, we found it to be too restrictive in many practical cases.

We take a more liberal approach, allowing pure methods to return newly allocated objects provided that does not cause any discrepancy between the dynamic execution and static verification. In the remainder of this section, we outline the analysis.

New attributes. The analysis requires the introduction of two new attributes. A reference-type pure method marked with attribute ResultNotNewlyAllocated says that the returned object has been allocated before the method was called. We encode this property as an additional postcondition

$$\textbf{ensures result} == \textbf{null} \vee \textbf{old}(h)[\textbf{result}, \mathit{allocated}];$$

where $\textbf{old}(h)$ refers to the value of the heap on entry to the method, and *allocated* is a special field that encodes whether or not the object has been allocated.

A method marked with the attribute NoReferenceComparison says that the method implementation does not perform any reference comparison (*i.e.*, use any operator == or != with reference-type operands). Two simple compile-time checks enforce this property: (1) the method may use reference comparison

only if one of the operands is the literal **null**; (2) for any call contained in the implementation, the callee must be marked with NoReferenceComparison.

By default, the attributes are not attached to methods, that is, methods may return newly allocated objects and may compare references.

Definition of allocating expressions. We introduce the notion of an *allocating expression* to describe an expression that may yield a newly allocated object. An expression e is considered to be allocating if and only if e is of reference type and is either (a) a constructor or method call where the callee is not marked with ResultNotNewlyAllocated; or (b) a composite expression with any allocating sub-expression.

The rationale behind (a) is self-evident; (b) is more subtle. Consider a field access "**this**.M().f" where method M is allocating, *i.e.* not marked with ResultNotNewlyAllocated. Then M might return a newly allocated object whose field f refers to a newly allocated object too. Our definition of allocating over-approximates the set of expressions that yield newly allocated objects; however, this keeps the analysis sound and simple.

Checks on specification expressions. Having defined allocating expressions, we can spell out the restrictions our analysis enforces on specifications: (1) reference comparisons may have at most one operand that is allocating; (2) if two or more parameters (possibly the receiver) of a method call are allocating, then the callee must be one that is marked with NoReferenceComparison.

Attribute inheritance. Finally, to ensure soundness in the presence of subtyping, we need a rule that forces an overriding method to be at least as restricted in what its implementation is allowed to do as the overridden counterpart. That is, if an overridden method is marked with one of the attributes then the overriding method must be marked with that attribute too.

Limitations. There are two main limitations of our design: methods need be annotated manually by users, and the syntactic nature of our analysis leads to over-approximation. We believe a more accurate analysis that goes beyond syntactic checks (*e.g.*, points-to analysis) could lessen these limitations: most annotations could be inferred and more methods could be annotated with NoReferenceComparison (since our requirements are not minimal).

On the other hand, as far as our experience went with the specification of Boogie, we only had to mark two methods with ResultNotNewlyAllocated (one in Boogie and one in mscorlib) and none with NoReferenceComparison. Furthermore, we did not encounter specifications that were rejected because of how over-approximative our analysis is.

Summary of contributions. In this section, we showed two new techniques that yield a simple and practical encoding and axiomatization of pure methods. A combination of simple syntactic measures can let the theorem prover off the hook for dealing with the quantifiers in formula (2): the identification of witness candidates, and the association of methods with numbers, most all of which can be inferred by a simple syntactic analysis. Our simple syntactic

allocating-expression analysis provides a plentiful solution to the tension between dynamic execution and static verification. These techniques do not rely on the specifics of the Spec# system, thus they can be adapted by other languages and program verifiers, too.

4 The Boogie Methodology

The remainder of the paper describes encodings that build on a particular state-of-the-art discipline of object invariants known as the *Boogie methodology* [1]. In this section, we review the parts of the methodology that are necessary for the understanding of the encodings.

Invariants and reentrancy. Any verification system needs to provide a way to specify and verify invariants. The basic idea of a condition declared to be an invariant is that by checking the condition at certain program points, one can safely assume that it holds at certain other program points. In the object-oriented setting, it is not clear where to check and assume the invariants of objects. For example, consider the simple approach of checking object invariants at the end of constructors, assuming them on entry to methods, and checking them again at the end of methods. This simple approach is not sound if a call from inside a method (where the object invariant may be temporarily broken) instigates a chain of (perhaps dynamically dispatched) calls that reenter a method of the object (where the simple approach said to assume invariants).

An approach that solves this problem is to explicitly track whether or not an object is in a state where its object invariant is certain to hold [1]. Following that approach, we say in this paper that an object is either *consistent*, which implies its object invariant holds, or *exposed*, which allows the object's fields to be updated and allows the object invariant to be violated. In this approach, known as the Boogie methodology, an appropriate precondition for a typical method is that the receiver object is consistent. The implementation then typically changes the state of the object from consistent to exposed, makes desired updates on the object's fields, and finally changes the state of the object back to consistent, at which point the object invariant is checked. Spec# has a special block statement, the **expose** statement, that performs the state change from consistent to exposed at the beginning of the block and back at the end of the block.

Aggregate objects. One other important thing in object-oriented specification and verification is the handling of aggregate objects. An aggregate object is a collection of separately allocated objects that together form one logical object. For example, an *Engine* may be an aggregate object that contains several *Cylinder* objects and a *FuelPump* object. A typical situation is then that a method of the Engine object, whose precondition says that the engine is consistent, calls methods on the Cylinder and FuelPump objects, whose preconditions say that those respective objects are consistent. The correctness of the Engine method thus relies on that the engine's cylinders and fuel pump are consistent, but

explicitly mentioning the consistency of those objects in the precondition of the Engine method would be a violation of information hiding.

An approach that solves this problem is structuring the heap according to a hierarchical *ownership* relation [4]. For example, one would specify that an engine is the owner of its cylinders and fuel pump. We say that the cylinders and fuel pump are representation objects, or just *rep* objects, of the engine object. Using the ownership relation, a verification approach can ensure that the consistency of an object implies the consistency of the objects it owns [1]. This addresses the typical situation outlined above, because it allows an Engine method to meet the preconditions of the calls it makes to various Cylinder and FuelPump methods.

For this approach to be sound, one needs to make sure that an object's state is changed from consistent to exposed only if the object's owner is already in the exposed state. Consequently, any method whose implementation exposes the receiver object needs to include in its precondition that the receiver object's owner is exposed. We say that an object is *committed* when its owner is consistent. In summary, an appropriate precondition for a typical method is that the receiver object is consistent and not committed.

In the sequel, we refer to the *ownership cone* of an object o to mean the set of objects that includes o (pictured on top of the cone), the rep objects of o (pictured one level below o), the rep objects of the rep objects (pictured yet another level below), and so on. That is, the ownership cone of o is a hierarchical collection of objects that includes o and all objects transitively owned by o.

Frame conditions. An important part of a method specification is its *frame condition*, which describes, as a postcondition, which locations in the heap the method may modify. Commonly, the frame condition is produced partly from a user-specified *modifies clause* and partly from rules prescribed by the methodology. For example, a standard rule is to allow fields of newly allocated objects to be modified.

A typical method implementation modifies the fields of the receiver object, the fields of its rep objects, the fields of their rep objects, and so on—that is, it modifies the heap locations in the ownership cone of the receiver object. We must therefore make it possible to include an object's entire ownership cone in a method's frame condition.

It is not possible to list all the locations of an ownership cone by name in the modifies clause, because doing so would violate information hiding. One approach is to instead use some encoding that yields the fields of all transitive rep objects, which can be done by quantifying over field names and using some transitive closure or reachability operator. However, experience shows that such a closure or reachability construct is hard to process well with an automatic theorem prover (*cf.* [9]).

The Boogie methodology takes a simple approach that over-approximates the heap locations that an implementation is able to modify: it says that, in addition to heap locations explicitly indicated in the modifies clause and heap locations

that belong to newly allocated objects, a method's frame condition includes the fields of all committed objects. More precisely, a modifies clause W gives rise to the following frame condition:

$$(\forall\, o, f \bullet\ h[o,f] = \mathbf{old}(h)[o,f] \vee (o,f) \in \mathbf{old}(W) \vee \neg\mathbf{old}(h)[o, \mathit{allocated}] \vee (o \text{ is committed in } \mathbf{old}(h)))\quad (3)$$

where o ranges over non-null object references, f over field names, and a heap location is a pair (o, f). This encoding has the advantage that it uses a simple quantification—it avoids transitive closure and reachability—and gives an implementation the license to modify fields of its transitive rep objects. One can also give an argument for why callers need not be adversely affected by the over-approximation [1].

5 Encoding Lightweight Read-Effects

In this section, we describe an encoding and reasoning technique that makes use of lightweight read-effect annotations of pure methods. By *lightweight* we mean that users need not prescribe precisely the effects but rather in an abstract, (over-)approximative way in terms of ownership cones.

In our attempts to run Boogie to verify Spec# code, we repeatedly find code patterns like the following, where M is a pure method:

```
if (o.M() != null) { r.P();  y = o.M().f; }
```

This code dereferences o.M() after checking that it is not null. One way to show that it does not throw a null-pointer exception is to prove that the two calls to o.M() return the same value. The basic verification support needed in such situations is knowing whether some heap changes (such as those performed by the call to r.P() in the example above) affect the value returned by a pure method.

Determining whether a heap change has any effect on the value returned by a pure method depends on what the pure method reads. We find it useful to consider three kinds of pure methods: (1) those that do not read any mutable part of the heap (called *state-independent* pure methods); (2) those that confine what they read to heap locations in the ownership cone of the receiver object (so-called *read-confined* pure methods, which are the most common kind of pure methods); and (3) those without any restrictions on what they might read (called *read-anything* methods). We require that every pure method be declared to be of one of these three read levels.

We do not go into details of how to enforce read restrictions. In our implementation of Spec#, we currently use a variation of the data-flow analysis prescribed by Sălcianu and Rinard [18].

Since a state-independent method M does not depend on the heap, we drop the heap argument from the signature of function $\#M$. This encoding makes it evident that the value returned by M in code patterns like the one above is not affected by heap changes.

Methods that can read anything are always potentially sensitive to changes in the heap, so we do not do anything extra for these methods.

For the most common kind of pure methods, the read-confined ones, we would like to provide an axiom that says, "if the values of the heap locations in the ownership cone of an object o are the same in two object stores h and k, then the return value of o.M() is the same in h and k". There are two problems with trying to supply such an axiom.

One problem is that describing, in the axiom, all heap locations in the receiver's ownership cone requires transitive closure or reachability, which makes it difficult to apply automatic theorem provers. The other problem is that even if the axiom mentions the entire ownership cone, frame condition (3) does not say anything about how the state of a committed object changes. Hence, a call like r.P() in the example above is viewed as possibly having an effect on the committed objects in the ownership cone of o. That is, with frame condition (3), an axiom that mentions the entire ownership cone is too weak to be useful.

To find a solution, we recall how the state of the ownership cone of an object can change in the Boogie methodology. The methodology enforces that a field s.x is updated only when s and all transitive owners of s are exposed. Consequently, an object's ownership cone remains unchanged through any time period during which the object remains consistent. We will seek to encode this fact by recording a *snapshot* of an object's ownership cone at the time the object transitions from exposed to consistent. The snapshot may become out-of-date when the object is in the exposed state, so we need to make sure to use snapshots only for consistent objects. Equipped with snapshots, which we represent by adding a field *snapshot* to every object (transparent to users), we obtain the following property:

$$\begin{array}{l}(\forall\, o, p, h, k \bullet\ (o \text{ is consistent in } h) \wedge (o \text{ is consistent in } k) \wedge \\ \qquad h[o, \mathit{snapshot}] = k[o, \mathit{snapshot}] \ \Rightarrow\ \ \#M(o, p, h) = \#M(o, p, k)\ )\end{array} \tag{4}$$

where o ranges over non-null object references, p ranges over the possible values of M's other parameters, and h and k range over heaps. Note the limited role of p in (4); this is due to the read-confinedness of the method, *i.e.* it may not read heap locations in the ownership cones of parameters. Axiom (4) lets us prove the example above—provided r.P() does not change o.snapshot, of course.

Encoding. To make the idea described here a reality in a checker, we need to design a suitable encoding for updating snapshot fields (to be used by the static verifier) and an appropriate formulation of axiom (4).

We update the snapshot field of an object o at the end of the constructor of o (when the object first becomes consistent) and at the end of **expose** blocks. We might consider the update of the field as an assignment statement like:

$$\text{o.snapshot} := (\text{o.x, o.y, o.z, } \ldots, \text{ o.r.snapshot, o.s.snapshot, } \ldots) \tag{5}$$

where the right-hand side is a tuple of the (non-snapshot) fields x, y, z, ... of o and the snapshots of the rep fields r, s, ... of o. This works if the right-hand side represents the entire ownership cone of o. But in Spec#, we allow an unbounded number of rep objects (à la Leino and Müller [12]), so it is not sufficient to include the snapshots of rep fields. Instead, we abstract over the actual snapshot

and assign to o.snapshot an arbitrary value satisfying property (4). Given that we postulate the property as an axiom, the assignment simply becomes "**havoc** o.snapshot" where **havoc** is a command that sets its l-value argument to an arbitrary value (satisfying postulated axioms).

Like for other fields, if a method has an effect on the snapshot field of a non-new, non-committed object, then it must account for the effect in the method's modifies clause. To make this easier on the programmer, our implementation implicitly adds **this**.snapshot to the modifies clause of every non-pure method. In addition, for any term explicitly mentioned in the modifies clause, say p.f.g where p is either **this** or another method parameter, our implementation implicitly adds p.snapshot and p.f.snapshot to the modifies clause.

As for axiom (4), we do not encode it the way we just showed it above, because quantifying over pairs of heaps does not give rise to good performance in the theorem prover. The axiom really states that, for any consistent object o and parameters p, $\#M(o, p, h)$ depends only on the reference o, the values p, and o.snapshot, that is, $\#M(o, p, h)$ is a function of o, p, and o.snapshot. So, we abstract again and introduce an uninterpreted function symbol $\#\#M$ and write the axiom simply as:

$$(\forall\, o, p, h \bullet\ (o \text{ is consistent in } h) \quad \Rightarrow \\ \#M(o, p, h) = \#\#M(o, p, h[o, \mathit{snapshot}])\ ) \tag{6}$$

This encoding solves the problem in the example above and it is the encoding we use in Boogie.

6 Preconditions and Frame Conditions for Pure Methods

This section discusses how to write preconditions for pure methods, motivating that a more relaxed condition is needed than the one used by the Boogie methodology. Then, we discuss what is an appropriate frame condition for pure methods and introduce axioms to regain some precision lost by the relaxed preconditions.

6.1 Consequences of the Standard Precondition

Let us take a closer look at a typical method, which will lead us to some considerations for the preconditions of pure methods. Consider the Engine class example mentioned in Section 4 and suppose it has the following method, which does the typical thing of calling a method on a rep object:

```
void RevUpEngine()
  requires this is consistent and not committed;
{ expose (this) { this.pump.IncreaseVolume(); } }
```

Let's review why this code needs the **expose** statement around the call. Typically, the form of the precondition of the called method, IncreaseVolume, is the same as that of the precondition of the caller, RevUpEngine: the receiver (here, pump) is consistent and not committed. To meet the precondition "not committed" of IncreaseVolume, it is necessary for method RevUpEngine to expose the engine object before invoking the method on the rep object.

Now, consider the situation where both the calling method and the called method are pure, for example where a pure GetRevStatus method calls a pure GetVolume method on the pump. In this situation, which occurs in practice, the pure caller method is not allowed to change the state of objects, thus it cannot expose the receiver object, and thus it cannot establish the called method's precondition that the rep object not be committed. To allow this situation, we need a more liberal precondition for pure methods. In particular, we would like to drop the part about the receiver not being committed. That part of the precondition was needed to support exposing and mutating an object, which pure methods are not allowed to do anyway.

In summary, an appropriate precondition for a pure method says that the receiver object is consistent, but says nothing about whether the object is committed or not, *i.e.*, whether the object's owner is consistent or exposed.

6.2 Frame Conditions of Pure Methods

It is clear that a pure method must have an empty modifies clause. However, it is not equally clear if the frame condition for a pure method is prescribed in the same way as it is for a mutating method.

Initially, we had thought of dropping the fourth disjunct of the frame condition (3) for pure methods. Such a design had seemed to make sense, because pure methods are not allowed to modify the committed objects in the cone of the receiver or other parameter. However, consider a pure method with the body "**return** (**new** T()).P();" where P is a method that mutates its receiver and the cone of the receiver. It seems reasonable to want to allow this code, because it operates only on newly allocated objects (the third disjunct). However, since both the T constructor and the P method include the fourth disjunct in *their* frame conditions (since they are mutating methods), it will be difficult to prove that M has no effect on the state of committed objects.

Instead, we propose keeping the fourth disjunct in the frame condition of pure methods. This gives pure method M the license to modify committed objects, thus it is allowed to call the T constructor and P method, whose frame conditions give them the license to modify committed objects. Despite having the license, we will see below that pure methods cannot actually modify committed objects.

The appropriate precondition for pure methods (which does not say whether or not the receiver is committed) and the frame condition of methods (which does not constrain the state of committed objects) interact. We discovered this bad interaction when we tried to verify code of the following form:

```
[Pure] int M()
{ int a = this.x;  int b = N();  assert a == this.x;  return b − a; }
```

where N is some pure method (whose frame condition, like the frame condition of all methods, includes the fourth disjunct). So, N is seen as having an arbitrary effect on the state of committed objects. This is a problem for the verification of the assert if the receiver is committed, which the appropriate precondition for pure methods (in particular, the precondition of M) allows it to be.

Intuitively, we can justify why the assert above will hold as follows. Any effect N has on **this**.x must happen when **this** is exposed. But **this** is consistent on entry to (M and) N, so N must at some point expose **this**. Exposing an object can only be done if its owner is already exposed. Thus, if **this** is committed on entry to N, then N must first expose the owner of **this**, and this same argument applies to the owner of **this**, and so on. Every committed object has some transitive owner that is consistent and not committed. In fact, that transitive owner is unique in any given state, let's call it the "first non-committed owner". So, if **this** is committed, then in order to expose **this**, one must start with exposing the first non-committed owner of **this**. This, according what we have described in Section 5, will have an irrevocable effect on the *snapshot* of that first non-committed owner. Since the first non-committed owner is both allocated and not committed, the only way its snapshot is allowed to be changed is if it is included in W, the set of modifiable locations of the method. Since pure methods have empty modifies clauses, we conclude that N does not have any effect on the snapshot of the first non-committed owner of **this**, and thus no effect on **this**.x.

We encode the idea of a first non-committed owner using a field *FirstNon-CommittedOwner* (*fnco*, for short), which has a meaningful value for any allocated, committed object. We do not want to give a precise axiomatization of fnco, because doing so would mean we need transitive closure or reachability. Instead, we just encode the properties we need, namely that fnco is allocated and not committed (which excludes it from the third and fourth disjuncts of the frame condition):

$$(\forall\, o, h \bullet\ h[o, \mathit{allocated}] \wedge (o \text{ is committed in } h) \quad \Rightarrow \\ h[o, \mathit{fnco}] \neq \mathbf{null} \wedge h[h[o, \mathit{fnco}], \mathit{allocated}] \wedge (h[o, \mathit{fnco}] \text{ not committed in } h)\ )$$

And we also need to say that a change to a field of a committed object requires a change in the snapshot of the object's first non-committed owner.

$$(\forall\, o, f, h \bullet\ h[o, \mathit{allocated}] \wedge (o \text{ is committed in } h) \quad \Rightarrow \\ (h[o, f] \text{ is a function of } o,\ f, \text{ and } h[h[o, \mathit{fnco}], \mathit{snapshot}]\ )\ )$$

Finally, o.fnco can change only if o.fnco.snapshot does, a property that we formalize into a postcondition assumption after each call.

With these axioms and assumptions, we are able to regain information about the values of fields and can prove the correctness of pure method M above.

Summary of contributions. We presented an axiom that exploits methods' read-confinedness, investigated what the proper precondition of pure methods is, and proposed axioms to regain information due to that precondition. Without these results, many code patterns often encountered in Spec# could not be handled.

7 Related Work and Conclusion

The work closest to ours on the sound axiomatization of pure methods is that of Darvas and Müller [6]. In fact, we follow all their proposals with adaptations required due to the different settings of our work. For example, our design is more liberal in allowing pure methods to return newly allocated objects.

Jacobs and Piessens [8] present a reasoning technique for confined methods that is very similar to our approach of using snapshots (Section 5). Their notion of confinedness is extended to method parameters too—which could be encoded in our settings as well with a slight adaption of axiom (6). They record their snapshots, which are represented as fields f_{state}, explicitly from the state fields of the rep objects. This works only if the rep objects can be enumerated syntactically (*cf.* (5)). However, it seems that they could use a havoc statement instead, like we do, to overcome that limitation.

Separation logic divides up the heap and keeps track of accessible regions of it. For example, the work of Parkinson and Bierman [16] can be used to encode the basic Boogie methodology. Read and write effects can be handled, but the approach has not been developed into a practical tool for a language like Spec#.

Kassios introduces a promising and powerful way to specify and reason about changes in the heap [10]. The technique is not tied to any particular methodology. We look forward to seeing it implemented in a tool.

In many ways, pure methods look like *model fields*, abstract variables whose values are defined in terms of more concrete fields. There are, however, some differences; for example, a model field has to be listed in the modifies clause of a method if the method changes its value. We would like to study the relation between pure methods and the model-field encoding by Leino and Müller [13].

Conclusion. We have presented encoding, axiomatization, and reasoning techniques for pure methods as implemented in the Spec# system. The techniques work with an automatic theorem prover and fit into a methodology for object invariants. Our work contributes a list of verification differences between ordinary and pure methods, an adaptation and extension the results on axiomatizing pure methods in [6], and an axiomatization of the special properties of read-confined pure methods that makes them insensitive to changes in other objects, an exploration of how to write preconditions and frame conditions for pure methods and how to provide a practical axiomatization to support these.

Acknowledgments. We thank Peter Müller for helpful discussions and Mike Barnett for discussions and his help with the implementation. The idea of taking a "snapshot" of an object's state is originally from Bart Jacobs. Diego Garbervetsky implemented the effects analysis in the Spec# system.

References

1. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *JOT*, 3(6):27–56, 2004.
2. M. Barnett, R. DeLine, B. Jacobs, B.-Y. E. Chang, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.
3. 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, 2005.
4. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, volume 33(10), pages 48–64. ACM, 1998.

5. D. R. Cok. Reasoning with specifications containing method calls and model fields. *JOT*, 4(8):77–103, October 2005.
6. Á. Darvas and P. Müller. Reasoning about method calls in interface specifications. *JOT*, 5(5):59–85, June 2006.
7. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Tech. Rep. HPL-2003-148, Systems Research Center, HP Labs, 2003.
8. B. Jacobs and F. Piessens. Verification of programs using inspector methods. In *Formal Techniques for Java-like Programs*, 2006.
9. R. Joshi. Extended static checking of programs with cyclic dependencies. Technical Note 1997-028, Digital Equipment Corporation Systems Research Center, 1997.
10. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, volume 4085 of *LNCS*, pages 268–283. Springer, 2006.
11. 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.
12. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, volume 3086 of *LNCS*, pages 491–516. Springer, 2004.
13. K. R. M. Leino and P. Müller. A verification methodology for model fields. In *ESOP*, volume 3924 of *LNCS*, pages 115–130. Springer, 2006.
14. C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, Jan.–Mar. 2004.
15. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
16. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM, 2005.
17. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In *ESOP*, volume 1576 of *LNCS*, pages 162–176. Springer, 1999.
18. A. Sălcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, volume 3385 of *LNCS*, pages 199–215. Springer, 2005.