Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic

Jan Smans, Bart Jacobs, and Frank Piessens

Katholieke Universiteit Leuven, Belgium

Abstract. The dynamic frames approach has proven to be a powerful formalism for specifying and verifying object-oriented programs. However, it requires writing and checking many frame annotations. In this paper, we propose a variant of the dynamic frames approach that eliminates the need to explicitly write and check frame annotations. Reminiscent of separation logic's frame rule, programmers write access assertions inside pre- and postconditions instead of writing frame annotations. From the precondition, one can then infer an upper bound on the set of locations writable or readable by the corresponding method. We implemented our approach in a tool, and used it to automatically verify several challenging programs, including subject-observer, iterator and linked list.

1 Introduction

Last year's distinguished paper at ECOOP, Regional Logic for Local Reasoning about Global Invariants [1], proposed Hoare-style proof rules for reasoning about dynamic frames in a Java-like language. In the dynamic frames approach [1–6], the programmer specifies upper bounds on the locations that can be read or written by a method in terms of expressions denoting sets of locations. To preserve information hiding, these expressions can involve dynamic frames, pure methods or ghost fields that denote sets of locations. A disadvantage of this approach is that frame annotations must be provided for each method, and that they must be checked explicitly at verification time.

This paper improves upon regional logic and other dynamic frames-based approaches in two ways: (1) method contracts are more concise and (2) fewer proof obligations must be discharged by the verifier. More specifically, we propose a variant of the dynamic frames approach inspired by separation logic that eliminates the need to explicitly write and check frame annotations. Instead, frame information is inferred from access assertions in pre- and postconditions. We have proven the soundness of our approach, implemented it in a verifier prototype and demonstrated its expressiveness by verifying several challenging examples from related work.

The remainder of this paper is structured as follows. In Section 2, we show how our approach solves the frame problem. Section 3 extends this solution with support for data abstraction. In Section 4, we sketch the soundness argument (for the complete proof, see [7]). Subclassing and inheritance are discussed in Section 5 . Finally, we discuss our experience with the verifier prototype, compare with related work, and conclude in Sections 6, 7 and 8.

2 Framing

To reason modularly about a method invocation, one should not rely on the callee's implementation, but only on its specification. For example, consider the code in Figure 1(b). To prove that the assertion at the end of the code snippet holds in every execution, one should only take into account *Cell*'s method contracts. However, the given contracts are too weak to prove the assertion. Indeed, *setX*'s implementation is allowed to change the state arbitrarily, as long as it ensures that **this**.x equals v on exit. In particular, the contract does not prevent $c_2.setX(10)$ from modifying $c_1.x$.

```
class Cell {
  int x;

  Cell()
    ensures this.x = 0;
  { this.x := 0; }

  void setX(int v)
    ensures this.x = v;
  { this.x := v; }
}
```

(a)

```
Cell c1 := new Cell();
c1.setX(5); //A

Cell c2 := new Cell();
c2.setX(10);

assert c1.x = 5;
```

(b)

Fig. 1. A class *Cell* and some client code.

To prove the assertion at the end of Figure 1(b), we must strengthen *Cell*'s method contracts. More specifically, the contracts should additionally specify an upper bound on the set of memory locations modifiable by the corresponding method. This problem is called the *frame problem.*

Various solutions to the frame problem have been proposed in the literature (see Section 7 for a detailed comparison). The solution proposed in this paper is as follows. A method may only access a memory location $o.f$ if it has permission to do so. More specifically, writing to or reading from a memory location $o.f$ requires $o.f$ to be *accessible*. Accessibility of $o.f$ is denoted **acc**$(o.f)$. Method implementations are not allowed to mention **acc**$(o.f)$. In particular, they are not permitted to branch over accessibility of a memory location. As a consequence, a location $o.f$ that was allocated before execution of a method m is only known to be accessible during execution of m if m's precondition requires accessibility of $o.f$. In other words, a method's precondition provides an upper bound on the set of memory locations modifiable by the corresponding method: a method can only modify an existing location $o.f$ if that location is required to be accessible by its precondition. As an example, consider the revised version of the class *Cell* of Figure 2. *setX* can only modify **this**.x, since its precondition only requires accessibility of **this**.x. Similarly, *Cell*'s constructor does not require access to any location, and can therefore only assign to fields of the new object.

```
class Cell {
  int x;

  Cell()
    ensures acc(this.x) ∧ this.x = 0;
  { this.x := 0; }

  void setX(int v)
    requires acc(this.x);
    ensures acc(this.x) ∧ this.x = v;
  { this.x := v; }
}
```

Fig. 2. A revised version of the class *Cell* from Figure 1(a).

The accessibility of a memory location can change over time. For example, when a new object is created, the fields of the new object become accessible. How does a method invocation affect the set of accessible memory locations? Since Java does not provide a mechanism for explicit deallocation and assertions can only mention allocated locations, it would be safe to assume that the set of accessible locations only grows across a method invocation. However, this assumption would rule out interesting specification patterns, where a method "captures" accessibility of a location. Furthermore, this assumption would break in the presence of concurrency, where accessibility of memory locations is passed on to other threads (cfr. [8, 9]). Therefore, we use the following rule instead: a memory location $o.f$ that is known to be accessible before a method invocation is still accessible after the invocation, if $o.f$ was not required to be accessible by the callee's precondition. On the other hand, a location $o.f$ that was required to be accessible by the callee's precondition is still accessible after the call only if the callee's postcondition ensures accessibility of $o.f$. In other words, **acc**$(o.f)$ in a precondition transfers permission to access $o.f$ from the caller to the callee, and vice versa **acc**$(o.f)$ in a postcondition returns that permission to the caller.

Given the new method contracts for *Cell* of Figure 2 together with the rules for framing outlined above, we can now prove the assertion at the end of Figure 1(b). Informally, the reasoning is as follows. At program location A, the postcondition of $c_1.setX(5)$ holds: $c_1.x$ is accessible and its value is 5. Since c_2's constructor does not require access to any location, it can modify neither the accessibility nor the value of any existing location. In particular, $c_1.x$ is still accessible and still holds 5. Similarly, the call $c_2.setX(10)$ only requires $c_2.x$ to be accessible, and hence $c_1.x$ is not affected. We may conclude that the assertion, $c_1.x = 5$, holds in any execution.

2.1 Formal Details

In the remainder of this section, we describe a small Java-like language with contracts. Secondly, we define the notion of valid program. Informally, a program π is valid if each method successfully verifies, i.e. if the verification conditions of π's methods are valid.

Language We describe the details of our verification approach with respect to the small Java-like language of Figure 3. A program consists of a number of classes and a main routine $\overline{s}$. Each class declares a number of fields and methods. For now, we consider only mutator methods. Each mutator method has a corresponding method body, consisting of a sequence of statements. A statement is either a field update, a variable declaration, a variable update, an object construction, a mutator invocation or an assert statement. In addition, a mutator method declares a method contract, consisting of two assertions: a precondition and a postcondition. An assertion is either **true**, an access assertion, a conjunction, a separating conjunction, an equality or a conditional assertion. A separating conjunction holds only if both conjuncts hold and the left and right-hand side demand access to disjoint parts of the heap. Both statements and assertions contain expressions. An expression is either a variable, a field read, or a constant (**null** or an integer constant).

$$
\begin{array}{lll}
\mathit{program} & ::= & \overline{\mathit{class}}\ \overline{s} \\
\mathit{class} & ::= & \mathbf{class}\ C\ \{\ \overline{\mathit{field}}\ \overline{\mathit{method}}\ \} \\
\mathit{field} & ::= & t\ f; \\
\mathit{method} & ::= & \mathit{mutator} \\
\mathit{mutator} & ::= & \mathbf{void}\ m(\overline{t\ x})\ \mathit{contract}\ \{\ \overline{s}\ \} \\
\mathit{contract} & ::= & \mathbf{requires}\ \phi;\ \mathbf{ensures}\ \phi; \\
t & ::= & \mathbf{int} \mid C \\
s & ::= & e.f := e;\ \mid t\ x;\ \mid x := e;\ \mid x := \mathbf{new}\ C;\ \mid e.m(\overline{e});\ \mid \mathbf{assert}\ e = e; \\
\phi & ::= & \mathbf{true} \mid \mathbf{acc}(e.f) \mid \phi \wedge \phi \mid \phi * \phi \mid e = e \mid e = e\ ?\ \phi : \phi \\
e & ::= & x \mid e.f \mid c
\end{array}
$$

Fig. 3. Syntax of a Java-like language with contracts.

We assume the usual syntactic sugar. In particular, a constructor

$$C(t_1\ x_1, \ldots, t_n\ x_n)\ \mathbf{requires}\ \phi_1;\ \mathbf{ensures}\ \phi_2;\ \{\ \overline{s}\ \}$$

is a shorthand for the mutator method

$$
\begin{array}{l}
\mathbf{void}\ \mathit{init}_C(t_1\ x_1, \ldots, t_n\ x_n) \\
\quad \mathbf{requires}\ \mathbf{acc}(f_1) * \ldots * \mathbf{acc}(f_n) * \phi_1;\ \mathbf{ensures}\ \phi_2; \\
\{\ \overline{s}\ \}
\end{array}
$$

where $f_1, \ldots, f_n$ are the fields of C. Accordingly, a constructor invocation $x :=$ **new** $C(e_1, \ldots, e_n)$; abbreviates $x := \mathbf{new}\ C;\ x.\mathit{init}_C(e_1, \ldots, e_n);$.

Verification We check the correctness of a program by generating verification conditions. The verification conditions are first-order formulas whose validity implies the correctness of the program. In our implementation, we rely on an SMT solver [10] to discharge the verification conditions automatically.

Logic We target a multi-sorted, first-order logic with equality. That is, a term τ is either a variable or a function application. A formula ψ is either *true*, *false*, a conjunction, a disjunction, an implication, a negation, an equality among terms or a quantification. The formula $\mathsf{ite}(\tau_1 = \tau_2, \psi_1, \psi_2)$ is a shorthand for $(\tau_1 = \tau_2 \Rightarrow \psi_1) \wedge (\tau_1 \neq \tau_2 \Rightarrow \psi_2)$. An application of a function g with arity 0 is denoted g instead of $g()$. Functions with arity 0 are called constants.

Each term in the logic has a corresponding sort. The sorts are the following: the sort of values, *val*, the sort of object references, *ref*, the sort of integers, *int*, the sort of heaps, *heap*, the sort of booleans, *bool*, the sort of sets of memory locations, *set*, the sort of field names, *fname*, and finally the sort of class names, *cname*. We omit sorts whenever they are clear from the context.

The signature of the logic consists of *built-in functions* and a number of *program-specific functions*. The built-in functions include the following:

function	sort
null	*ref*
emptyset	*set*
singleton	$\mathit{ref} \times \mathit{fname} \rightarrow \mathit{set}$
intersect	$\mathit{set} \times \mathit{set} \rightarrow \mathit{set}$
union	$\mathit{set} \times \mathit{set} \rightarrow \mathit{set}$
contains	$\mathit{ref} \times \mathit{fname} \times \mathit{set} \rightarrow \mathit{bool}$
select	$\mathit{heap} \times \mathit{ref} \times \mathit{fname} \rightarrow \mathit{val}$
store	$\mathit{heap} \times \mathit{ref} \times \mathit{fname} \times \mathit{val} \rightarrow \mathit{heap}$
allocated	$\mathit{ref} \times \mathit{heap} \rightarrow \mathit{bool}$
allocate	$\mathit{ref} \times \mathit{heap} \rightarrow \mathit{heap}$
ok	$\mathit{heap} \times \mathit{set} \rightarrow \mathit{bool}$
succ	$\mathit{heap} \times \mathit{set} \times \mathit{heap} \times \mathit{set} \rightarrow \mathit{bool}$

In addition to the built-in functions, the logic contains a number of program-specific functions. In particular, the logic includes a constant C with sort *cname* for each class C and a constant f with sort *fname* for each field f in the program text. In Section 3, we will introduce additional program-specific functions.

Interpretation We interpret the functions using the interpretation $\mathfrak{I}$. The interpretation of the built-in functions is as expected. More specifically, *null* is interpreted as the constant **null**. The functions *emptyset*, *singleton*, *union*, *intersect*, and *contains* are interpreted as their mathematical counterpart. We abbreviate applications of these functions by their mathematical notation. The function $\mathit{select}(h, o, f)$ corresponds to applying h to (o, f). Accordingly, $\mathit{store}(h, o, f, v)$ corresponds to an update of the function h at location (o, f) with v. We abbreviate $\mathit{select}(h, o, f)$ as $h(o, f)$ and $\mathit{store}(h, o, f, v)$ as $h[(o, f) \mapsto v]$. $\mathit{ok}(h, a)$ denotes that the state with heap h and access set a is well-formed. Well-formedness implies that both the access set a and the range of the heap h contain only allocated objects. $succ(h, a, h', a')$ states that the state with heap h' and access set a' is a successor of the state with heap h and access set a. Successors of well-formed states are well-formed. Furthermore, a successor state has more allocated locations than its predecessor. We interpret the built-in constant f as the field name f and the constant C as the class name C.

Theory We assume that the theory $\Sigma_{prelude}$ (incompletely) axiomatizes the built-in functions. That is, $\mathfrak{I}$ is a model for $\Sigma_{prelude}$: $\mathfrak{I} \models \Sigma_{prelude}$. $\Sigma_{prelude}$ may for instance contain a subtheory which axiomatizes the set functions. For example, in our verifier prototype the prelude includes an axiom that encodes that the empty set contains no locations:

$$\forall o, f \bullet (o, f) \notin emptyset$$

For now, we assume that Σ_π, the theory for verifying mutator methods and the main routine, equals $\Sigma_{prelude}$.

statements	verification condition
$e_1.f := e_2;\ \overline{s}$	$\mathsf{Df}(e_1) \land \mathsf{Df}(e_2) \land (\mathsf{Tr}(e_1), f) \in a \land$
	$\mathsf{vc}(\overline{s}, \psi)[h[(\mathsf{Tr}(e_1), f) \mapsto \mathsf{Tr}(e_2)]/h]$
$t\ x;\ \overline{s}$	$\forall x \bullet \mathsf{vc}(\overline{s}, \psi)$
$x := e;\ \overline{s}$	$\mathsf{Df}(e) \land \mathsf{vc}(\overline{s}, \psi)[\mathsf{Tr}(e)/x]$
$x :=$ **new** $C;\ \overline{s}$	$\forall y \bullet y \neq null \land \neg allocated(y, h) \Rightarrow$
	$\mathsf{vc}(\overline{s}, \psi)[y/x, (a \cup \{(y, f_1), \ldots, (y, f_n)\})/a, allocate(y, h)/h]$
	where $f_1, \ldots, f_n$ are the fields of C
$e_0.m(e_1, \ldots, e_n);\ \overline{s}$	$\mathsf{Df}(e_0) \land \ldots \land \mathsf{Df}(e_n) \land \mathsf{Tr}(e_0) \neq null \land \mathsf{Tr}(P) \land$
	$(\forall h', a' \bullet$
	$\quad succ(h, a, h', a') \land$
	$\quad \mathsf{Tr}(Q)[h'/h, a'/a] \land$
	$\quad (\forall o, f \bullet (o, f) \in a \setminus \mathsf{R}(P) \Rightarrow (o, f) \in a' \land h(o, f) = h'(o, f)) \land$
	$\quad (\forall o, f \bullet (o, f) \in \mathsf{R}(Q)[h'/h, a'/a] \setminus \mathsf{R}(P) \Rightarrow (o, f) \notin a)$
	$\quad \Rightarrow$
	$\quad \mathsf{vc}(\overline{s}, \psi)[h'/h, a'/a])$
	where C is the type of e_0,
	$x_1, \ldots, x_n$ are the parameters of $C.m$,
	P is $\mathsf{mpre}(C, m)[e_0/\mathbf{this}, e_1/x_1, \ldots, e_n/x_n]$ and
	Q is $\mathsf{mpost}(C, m)[e_0/\mathbf{this}, e_1/x_1, \ldots, e_n/x_n]$
assert $e_1 = e_2;\ \overline{s}$	$\mathsf{Df}(e_1 = e_2) \land \mathsf{Tr}(e_1 = e_2) \land \mathsf{vc}(\overline{s}, \psi)$
nil	ψ

Fig. 4. Verification conditions (vc) of statements with respect to postcondition ψ.

Verification Conditions We check the correctness of a program by generating verification conditions. The verification conditions for each statement are shown in Figure 4. The free variables of $\mathsf{vc}(\overline{s}, \psi)$ are h, a, and the free variables of ψ and $\overline{s}$. The variable h denotes the heap, while the variable a denotes the set of accessible locations. Tr and Df denote the translation and respectively the definedness of expressions and assertions (shown in Figure 5). $\mathsf{mpre}(C, m)$ and $\mathsf{mpost}(C, m)$ respectively denote the pre- and postcondition of the method $C.m$.

expression	Tr	Df
x	x	$true$
$e.f$	$h(\mathsf{Tr}(e), f)$	$\mathsf{Df}(e) \wedge (\mathsf{Tr}(e), f) \in a$
c	c	$true$
true	$true$	$true$
$\mathbf{acc}(e.f)$	$(\mathsf{Tr}(e), f) \in a$	$\mathsf{Df}(e) \wedge \mathsf{Tr}(e) \neq null$
$\phi_1 \wedge \phi_2$	$\mathsf{Tr}(\phi_1) \wedge \mathsf{Tr}(\phi_2)$	$\mathsf{Df}(\phi_1) \wedge (\mathsf{Tr}(\phi_1) \Rightarrow \mathsf{Df}(\phi_2))$
$\phi_1 * \phi_2$	$\mathsf{Tr}(\phi_1 \wedge \phi_2) \wedge (\mathsf{R}(\phi_1) \cap \mathsf{R}(\phi_2) = \emptyset)$	$\mathsf{Df}(\phi_1 \wedge \phi_2)$
$e_1 = e_2$	$\mathsf{Tr}(e_1) = \mathsf{Tr}(e_2)$	$\mathsf{Df}(e_1) \wedge \mathsf{Df}(e_2)$
$e_1 = e_2 \; ? \; \phi_1 : \phi_2$	$\mathsf{ite}(\mathsf{Tr}(e_1 = e_2), \mathsf{Tr}(\phi_1), \mathsf{Tr}(\phi_2))$	$\mathsf{ite}(\mathsf{Tr}(e_1 = e_2), \mathsf{Df}(\phi_1), \mathsf{Df}(\phi_2))$

Fig. 5. Translation (Tr) and definedness (Df) of expressions and assertions.

The first core ingredient of our approach is that a method can only access a memory location if it has permission to do so. To enforce this restriction, the verification condition for field update checks that the assignee is in the access set a. Similarly, a field read $o.f$ is only well-defined if $o.f$ is an element of a.

assertion	R
true	$\emptyset$
$\mathbf{acc}(e.f)$	$\{(\mathsf{Tr}(e), f)\}$
$\phi_1 \wedge \phi_2$	$\mathsf{R}(\phi_1) \cup \mathsf{R}(\phi_2)$
$\phi_1 * \phi_2$	$\mathsf{R}(\phi_1 \wedge \phi_2)$
$e_1 = e_2$	$\emptyset$
$e_1 = e_2 \; ? \; \phi_1 : \phi_2$	$\mathsf{ite}(\mathsf{Tr}(e_1 = e_2), \mathsf{R}(\phi_1), \mathsf{R}(\phi_2))$

Fig. 6. Required access set (R) of assertions.

The second core ingredient of our approach is that we deduce frame information from a callee's precondition. More specifically, a callee can only read or modify an existing location $o.f$ if its precondition demands access to $o.f$. A naive, literal encoding of this property does not lead to good performance with automatic theorem provers. In particular, the combination of the literal encoding and our approach for data abstraction of Section 3 yields verification conditions that are too hard for those provers. Therefore, we propose a slightly different encoding. More specifically, we syntactically infer from the callee's precondition a *required access set*, i.e. a term denoting the set of memory locations required to be accessible by the precondition. The definition of required access set (R) of an assertion is shown in Figure 6. The subformula

$$\forall o, f \bullet (o, f) \in a \setminus \mathsf{R}(P) \Rightarrow (o, f) \in a' \wedge h(o, f) = h'(o, f)$$

in the verification condition of method invocation encodes the property that all locations $o.f$ that are accessible to the callee and that were not in the required access set of the precondition remain accessible and retain their value. Note that this is a *free* postcondition: callers can assume the postcondition holds, but it is not necessary to explicitly prove the postcondition when verifying the method's implementation (see Definition 1). In addition to the "free modifies" clause, callers may assume a second free postcondition, the swinging pivot property:

$$\forall o, f \bullet (o, f) \in \mathsf{R}(Q)[h'/h, a'/a] \setminus \mathsf{R}(P) \Rightarrow (o, f) \notin a$$

The swinging pivot property states that all locations required to be accessible by the postcondition are either required to be accessible by the precondition or are not accessible to the callee. In Section 3, this property will be crucial to ensure disjointness.

A program is *valid* (Definition 3) if it successfully verifies. More specifically, a valid program only contains valid methods and has a valid main routine. A mutator is valid (Definition 1) if both its pre- and postcondition are well-defined assertions and if its body satisfies the method contract. A method body $\overline{s}$ satisfies the contract if the postcondition holds after executing $\overline{s}$, whenever execution starts in a state satisfying the precondition. The main routine is valid (Definition 2) if it satisfies the contract **requires true**; **ensures true**;. Executions of valid programs never deference null and assert statements never fail. We outline a proof of this property in Section 4.

Definition 1. *A mutator method*

$$\textbf{void } m(t_1\ x_1, \ldots, t_k\ x_k) \textbf{ requires } \phi_1;\ \textbf{ensures } \phi_2;\ \{\ \overline{s}\ \}$$

is valid *if all of the following hold:*

- *The precondition is well-defined and the postcondition is well-defined, provided the precondition holds.*

$$\Sigma_\pi \vdash \forall h, a, h', a', \mathit{this}, x_1, \ldots, x_k \bullet \mathit{ok}(h, a) \wedge \mathit{succ}(h, a, h', a') \wedge \mathit{this} \neq \mathit{null} \wedge \\ \Downarrow \\ \mathsf{Df}(\phi_1) \wedge (\mathsf{Tr}(\phi_1) \Rightarrow \mathsf{Df}(\phi_2)[h'/h, a'/a])$$

- *The method body satisfies the method contract.*

$$\Sigma_\pi \vdash \forall h, a, \mathit{this}, x_1, \ldots, x_k \bullet \mathit{ok}(h, a) \wedge \mathit{this} \neq \mathit{null} \wedge \mathsf{Tr}(\phi_1) \Rightarrow \mathsf{vc}(\overline{s}, \mathsf{Tr}(\phi_2))$$

Definition 2. *The main routine $\overline{s}$ is* valid *if the following holds:*

$$\Sigma_\pi \vdash \forall h, a \bullet \mathit{ok}(h, a) \Rightarrow \mathsf{vc}(\overline{s}, \mathit{true})$$

Definition 3. *A program π is* valid *(denoted* $\mathsf{valid}(\pi)$*) if all mutator methods and the main routine are valid.*

3 Data Abstraction

Data abstraction is crucial in the construction of modular programs, since it ensures that internal changes in one module do not propagate to other modules. However, the class *Cell* of Figure 2 and its specifications were not written with data abstraction in mind. More specifically, (1) client code must directly access the field x to query a *Cell* object's internal state and (2) *Cell*'s method contracts are not implementation-independent as they mention the internal field x. Any change to *Cell*'s implementation, such as renaming x to y, would break or at least oblige us to reconsider the correctness of client code.

```
class Cell {
  int x;

  Cell()
    ensures valid() ∧ getX() = 0;
  { this.x := 0; }

  void setX(int v)
    requires valid();
    ensures valid() ∧ getX() = v;
  { this.x := v; }

  predicate bool valid()
  { return acc(this.x); }

  pure int getX()
    requires valid();
  { return this.x; }

  void swap(Cell c)
    requires valid() ∗ c ≠ null ∧ c.valid();
    ensures valid() ∗ c.valid();
    ensures getX() = old(c.getX());
    ensures c.getX() = old(getX());
  { int i := x; x := c.getX(); c.setX(i); }
}
```

(a)

```
Cell c1 := new Cell();
c1.setX(5); //A

Cell c2 := new Cell();
c2.setX(10);

assert c1.getX() = 5;
```

(b)

Fig. 7. A revised version of class *Cell* with data abstraction.

Developers typically solve issue (1) by adding "getters" to their classes. For example, the class *Cell* of Figure 7(a) defines a method *getX* to query a *Cell*'s internal state. The method is marked **pure** to indicate it does not have side-effects. As shown in Figure 7(b), the assertion of Figure 1(b) can now be rephrased in terms of *getX*.

To complete the decoupling between *Cell*'s implementation and client code, we should also solve issue (2) and make *Cell*'s method contracts implementation-independent. In this paper, we solve the latter issue by allowing getters to be used inside specifications. That is, we allow the effect of one method to be specified in terms of other methods. For example, the behavior of $setX$ in Figure 7(a) is described in terms of its effect on $getX$.

In this paper, methods used within contracts are called *pure methods*. We distinguish two kinds of pure methods: normal pure methods (annotated with **pure**) and predicates (annotated with **predicate**). A pure method's body consists of a single return statement, returning either an expression (in case of a normal pure method) or an assertion (in case of a predicate). That is, a normal pure method abstracts over an expression, while a predicate abstracts an assertion. Since assertions and expressions are side-effect free, execution of a pure method never modifies the state. Since we disallow mentioning assertions inside method bodies, predicates can only be called from contracts and from the bodies of predicates. Furthermore, predicates are not allowed to have preconditions. In our running example, both $getX$ and $valid$ are pure methods. The former is a normal pure method, while the latter is a predicate. Predicates are typically used to represent invariants and to abstract over accessibility of memory locations.

To prove the assertion at the end of Figure 7(b), one must show that c_2's constructor and $c_2.setX(10)$ do not affect the return value of $c_1.getX()$. In other words, it suffices to show that the locations modified by those statements is disjoint from the set of locations that $c_1.getX()$ depends on. But how can we determine which locations influence the return value of $getX$? The answer is simple.

We can deduce from the precondition of a normal pure method an upper bound on the set of locations readable by that method: a pure method p can only read $o.f$ if p's precondition requires $o.f$ to be accessible. In other words, the return value of a normal pure method only depends on locations required to be accessible by its precondition. A predicate does not have a precondition, so what locations does its return value depend on? We say a predicate is *self-framing*. That is, the return value of a predicate q only depends on locations that q itself requires to be accessible.

Given these properties of pure methods, we can now prove the assertion at the end of Figure 7(b). Informally, the reasoning is as follows. At program location A, the postcondition of $c_1.setX(5)$ holds: $c_1.valid()$ is true and $c_1.getX()$ returns 5. Because c_2's constructor does not require access to any existing location, it can only modify fresh locations (i.e. c_2's fields and fields of objects allocated within the constructor itself). Since $c_1.valid()$ only requires access to non-fresh locations, both its own return value and the return value of $c_1.getX()$ are not affected by c_2's constructor. In addition, the set of memory locations required to be accessible by $c_1.valid()$ is disjoint from the set of locations required to be accessible by $c_2.valid()$, since the latter set only contains fresh locations (follows from the swinging pivot property). $c_2.setX()$ can only modify locations covered by $c_2.valid()$. The latter set of locations is disjoint from $c_1.valid()$, hence the return values of $c_1.\mathit{valid}()$ and $c_1.\mathit{getX}()$ are not affected by $c_2.\mathit{setX}(10)$. We may conclude that the assertion, $c_1.\mathit{getX}() = 5$, holds in any execution.

To illustrate the use of the separating conjunction, consider the method *swap* of Figure 7(a). *swap*'s precondition requires that the receiver and c are "separately" valid, i.e. that both $\mathbf{this}.\mathit{valid}()$ and $c.\mathit{valid}()$ hold and that the set of locations required to be accessible by $\mathbf{this}.\mathit{valid}()$ is disjoint from the set of locations required to be accessible by $c.\mathit{valid}()$. If we would have used a normal conjunction instead of a separating conjunction, we would not be able to prove $c.\mathit{valid}()$ holds after the assignment to x. In particular, the separating conjunction ensures that $c.\mathit{valid}()$ does not depend on $\mathbf{this}.x$.

3.1 Formal Details

Language We extend the language of Figure 3 with normal pure methods (typically denoted as p) and predicates (typically denoted as q) as shown in Figure 8. Accordingly, we add predicate invocations to the assertion language and normal pure method invocations to the expression language.

$$\begin{array}{lll}
\mathit{method} & ::= & \ldots \mid \mathit{predicate} \mid \mathit{pure} \\
\mathit{predicate} & ::= & \mathbf{predicate\ bool}\ q(\overline{t\ x})\ \{\ \mathbf{return}\ \phi;\ \} \\
\mathit{pure} & ::= & \mathbf{pure}\ t\ p(\overline{t\ x})\ \mathit{contract}\ \{\ \mathbf{return}\ e;\ \} \\
\phi & ::= & \ldots \mid e.q(\overline{e}) \\
e & ::= & \ldots \mid e.p(\overline{e})
\end{array}$$

Fig. 8. An extension of the language of Figure 3 with pure methods.

To ensure consistency of the encoding of pure methods, we enforce that pure methods terminate by syntactically checking that a pure method p only calls pure methods defined before p in the program text. We discuss this restriction together with more liberal solutions for ensuring consistency in Section 4.

Verification

Logic A standard technique in verification is to represent pure methods as functions in the verification logic [11, 12]. More specifically, for a normal pure method $C.p$ with parameters $t_1\ x_1, \ldots, t_n\ x_n$ and return type t, the verification logic includes a function $C.p$ with sort $\mathit{heap} \times \mathit{set} \times \mathit{ref} \times \mathsf{sort}(t_1) \times \ldots \times \mathsf{sort}(t_n) \to \mathsf{sort}(t)$, where sort maps a type to its corresponding sort. Similarly, for each predicate $C.q$ with parameters $t_1\ x_1, \ldots, t_n\ x_n$, the logic includes a function $C.q$ with sort $\mathit{heap} \times \mathit{set} \times \mathit{ref} \times \mathsf{sort}(t_1) \times \ldots \times \mathsf{sort}(t_n) \to \mathit{bool}$ and a function $C.q_{\mathsf{FP}}$ with sort $\mathit{heap} \times \mathit{set} \times \mathit{ref} \times \mathsf{sort}(t_1) \times \ldots \times \mathsf{sort}(t_n) \to \mathit{set}$. The latter function, $C.q_{\mathsf{FP}}$, is called q's footprint function.

An invocation of a pure method is encoded in the verification logic as an application of the corresponding function. For example, the postcondition of *setX* of Figure 7(a) is encoded as $\mathit{Cell.valid}(h, a, \mathit{this}) \land \mathit{Cell.getX}(h, a, \mathit{this}) = v$.

Interpretation We extend $\mathfrak{I}$ to these new program-specific functions as follows. For each normal pure method $C.p$ and for all heaps H, access sets A and values $v_0, \ldots, v_n$, $\mathfrak{I}(C.p)(H, A, v_0, \ldots, v_n)$ equals v, if evaluation of $v_0.p(v_1, \ldots, v_n)$ terminates and yields value v. Otherwise, $\mathfrak{I}(C.p)(H, A, v_0, \ldots, v_n)$ equals the default value of the method's return type. The interpretation of predicates and footprint functions is similar (see [7]).

Theory The behavior of a pure method is encoded via several axioms. Each normal pure method p has a corresponding axiomatization Σ_p, consisting of an implementation and a frame axiom. More specifically, the axioms corresponding to the normal pure method

$$\textbf{pure}\ t\ p(t_1\ x_1, \ldots, t_k\ x_k)\ \textbf{requires}\ \phi_1;\ \textbf{ensures}\ \phi_2;\ \{\ \textbf{return}\ e;\ \}$$

are the following:

– **Implementation axiom**. The implementation axiom relates the function symbol $C.p$ to the pure method's implementation: applying the function equals evaluating the method body, provided the precondition holds.

$$\begin{gathered}\forall h, a, \mathit{this}, x_1, \ldots, x_k \bullet \mathit{ok}(h, a) \wedge \mathit{this} \neq \mathit{null} \wedge \mathsf{Tr}(\phi_1) \\ \Downarrow \\ C.p(h, a, \mathit{this}, x_1, \ldots, x_k) = \mathsf{Tr}(e)\end{gathered}$$

– **Frame axiom**. The frame axiom encodes the property that a pure method only depends on locations in the required access set of its precondition. That is, the return value of p is the same in two states, if locations in the required access set of the precondition have the same value in both heaps.

$$\begin{gathered}\forall h_1, a_1, h_2, a_2, \mathit{this}, x_1, \ldots, x_k \bullet \\ \mathit{ok}(h_1, a_1) \wedge \mathit{ok}(h_2, a_2) \wedge \mathit{this} \neq \mathit{null} \wedge \\ \mathsf{Tr}(\phi_1)[h_1/h, a_1/a] \wedge \mathsf{Tr}(\phi_1)[h_2/h, a_2/a] \wedge \\ (\forall o, f \bullet (o, f) \in \mathsf{R}(\phi_1)[h_1/h, a_1/a] \Rightarrow (o, f) \in a_2 \wedge h_1(o, f) = h_2(o, f)) \\ \Downarrow \\ C.p(h_1, a_1, \mathit{this}, x_1, \ldots, x_k) = C.p(h_2, a_2, \mathit{this}, x_1, \ldots, x_k)\end{gathered}$$

Each predicate q has a corresponding axiomatization Σ_q, consisting of an implementation axiom, frame axiom, footprint implementation axiom, footprint frame axiom and a footprint allocated axiom. More specifically, the axioms corresponding to the predicate

$$\textbf{predicate bool}\ q(t_1\ x_1, \ldots, t_k\ x_k)\ \{\ \textbf{return}\ \phi;\ \}$$

are the following:

– **Implementation axiom**. The implementation axiom relates q's function symbol to its implementation.

$$\begin{gathered}\forall h, a, \mathit{this}, x_1, \ldots, x_k \bullet \mathit{ok}(h, a) \wedge \mathit{this} \neq \mathit{null} \\ \Downarrow \\ C.q(h, a, \mathit{this}, x_1, \ldots, x_k) = \mathsf{Tr}(\phi)\end{gathered}$$

– **Frame axiom**. The frame axiom encodes the property that a predicate is self-framing.

$$\begin{gathered}\forall h_1, a_1, h_2, a_2, \mathit{this}, x_1, \ldots, x_k \bullet \\ \mathit{ok}(h_1, a_1) \wedge \mathit{ok}(h_2, a_2) \wedge \mathit{this} \neq \mathit{null} \wedge \\ C.q(h_1, a_1, \mathit{this}, x_1, \ldots, x_k) \wedge \\ (\forall o, f \bullet (o, f) \in C.q_{\mathsf{FP}}(h_1, a_1, \mathit{this}, x_1, \ldots, x_k) \Rightarrow (o, f) \in a_2 \wedge h_1(o, f) = h_2(o, f)) \\ \Downarrow \\ C.q(h_2, a_2, \mathit{this}, x_1, \ldots, x_k)\end{gathered}$$

– **Footprint implementation axiom**. The footprint implementation axiom relates the function symbol $C.q_{\mathsf{FP}}$ to the required access set of the body of the predicate.

$$\begin{gathered}\forall h, a, \mathit{this}, x_1, \ldots, x_k \bullet \mathit{ok}(h, a) \wedge \mathit{this} \neq \mathit{null} \wedge C.q(h, a, \mathit{this}, x_1, \ldots, x_k) \\ \Downarrow \\ C.q_{\mathsf{FP}}(h, a, \mathit{this}, x_1, \ldots, x_k) = \mathsf{R}(\phi)\end{gathered}$$

– **Footprint frame axiom**. The footprint frame axiom encodes the property that a footprint function frames itself, provided the corresponding predicate holds.

$$\begin{gathered}\forall h_1, a_1, h_2, a_2, \mathit{this}, x_1, \ldots, x_k \bullet \\ \mathit{ok}(h_1, a_1) \wedge \mathit{ok}(h_2, a_2) \wedge \mathit{this} \neq \mathit{null} \wedge \\ C.q(h_1, a_1, \mathit{this}, x_1, \ldots, x_k) \wedge C.q(h_2, a_2, \mathit{this}, x_1, \ldots, x_k) \wedge \\ (\forall o, f \bullet (o, f) \in C.q_{\mathsf{FP}}(h_1, a_1, \mathit{this}, x_1, \ldots, x_k) \Rightarrow (o, f) \in a_2 \wedge h_1(o, f) = h_2(o, f)) \\ \Downarrow \\ C.q_{\mathsf{FP}}(h_1, a_1, \mathit{this}, x_1, \ldots, x_k) = C.q_{\mathsf{FP}}(h_2, a_2, \mathit{this}, x_1, \ldots, x_k)\end{gathered}$$

– **Footprint accessible axiom**. The footprint accessible axiom states that a predicate footprint only contains accessible locations, provided the predicate itself holds.

$$\begin{gathered}\forall h, a, \mathit{this}, x_1, \ldots, x_k \bullet \\ \mathit{ok}(h, a) \wedge \mathit{this} \neq \mathit{null} \wedge C.q(h, a, \mathit{this}, x_1, \ldots, x_k) \\ \Downarrow \\ C.q_{\mathsf{FP}}(h, a, \mathit{this}, x_1, \ldots, x_k) \subseteq a\end{gathered}$$

We redefine Σ_π as $\Sigma_{prelude} \cup \bigcup_{p \in \pi} \Sigma_p$. That is, Σ_π is the union of the axioms for the built-in functions and the axioms of each pure method in π. Moreover, we define Σ_{p*} as the axiomatization of all pure methods defined before p in the program text. Note that Σ_{p*} does not include Σ_p.

Verification Conditions To support data abstraction, we added pure methods and pure method invocation to our language. Figure 9 extends the table of Figure 5 with invocations of pure methods. In particular, pure methods are encoded as functions in the verification logic. An invocation of a pure method is well-defined if the arguments are well-defined and the receiver is not null. In

expression	Tr	Df
$e_0.p(e_1,\ldots,e_n)$	$C.p(h,a,\mathsf{Tr}(e_0),\ldots,\mathsf{Tr}(e_n))$	$\mathsf{Df}(e_0) \wedge \ldots \wedge \mathsf{Df}(e_n) \wedge \mathsf{Tr}(e_0) \neq \mathit{null} \wedge$ $\mathsf{Tr}(\mathsf{mpre}(C,p)[e_0/\mathbf{this}, e_1/x_1,\ldots,e_n/x_n])$
$e_0.q(e_1,\ldots,e_n)$	$C.q(h,a,\mathsf{Tr}(e_0),\ldots,\mathsf{Tr}(e_n))$	$\mathsf{Df}(e_0) \wedge \ldots \wedge \mathsf{Df}(e_n) \wedge \mathsf{Tr}(e_0) \neq \mathit{null}$

Fig. 9. Translation (Tr) and definedness (Df) of pure method invocations.

addition, the precondition must hold for a normal pure method invocation to be well-defined.

In this section, we added a new kind of assertion, namely predicate method invocation. What is the required access set of such an assertion? One solution would be to define the required access set of a predicate invocation as the required access set of the predicate's body. However, such a definition would expose implementation details to client code. For example, the required access set of the precondition of *getX* of Figure 7(a) would be the singleton $\{(\mathit{this}, x)\}$. Yet, this is just a detail of the current implementation, and client code should not rely on it. Instead, we propose introducing an extra layer of indirection. More specifically, as shown in Figure 10 the required access set of a predicate invocation is an application of the footprint function.

assertion	R
$e_0.q(e_1,\ldots,e_n)$	$C.q_{\mathsf{FP}}(h,a,\mathsf{Tr}(e_0),\ldots,\mathsf{Tr}(e_n))$

Fig. 10. Required access set (R) of predicate instances.

We redefine the notion of valid program. More specifically, for a program to be valid, we now additionally require that all pure methods are valid (Definition 6). Informally, a pure method is valid if its body and contract are well-defined (Definitions 4 and 5). Note that a pure method p is not verified with respect to the theory Σ_π but with respect to $\Sigma_{prelude} \cup \Sigma_{p*}$. That is, during verification of a pure method, one can only assume that the prelude axioms and axioms of pure methods defined before p in the program text hold.

Definition 4. *A predicate*

$$\textbf{predicate bool } q(t_1\ x_1,\ldots,t_k\ x_k)\ \{\ \textbf{return } \phi;\ \}$$

is valid if its body is a well-defined assertion:

$$\Sigma_{prelude} \cup \Sigma_{q*} \vdash \forall h, a, \mathit{this}, x_1, \ldots, x_k \bullet ok(h,a) \wedge \mathit{this} \neq \mathit{null} \Rightarrow \mathsf{Df}(\phi)$$

Definition 5. *A pure method*

$$\textbf{pure } t\ p(t_1\ x_1,\ldots,t_k\ x_k)\ \textbf{requires } \phi_1;\ \{\ \textbf{return } e;\ \}$$

is valid if its precondition is well-defined and its body is well-defined, provided the precondition holds:

$$\begin{gathered}\Sigma_{prelude} \cup \Sigma_{p*} \vdash \forall h, a, \mathit{this}, x_1, \ldots, x_k \bullet \\ ok(h, a) \wedge \mathit{this} \neq \mathit{null} \Rightarrow \mathsf{Df}(\phi_1) \wedge (\mathsf{Tr}(\phi_1) \Rightarrow \mathsf{Df}(e))\end{gathered}$$

Definition 6. *A program π is* valid *(denoted* $\mathsf{valid}(\pi)$*) if all methods (both pure and mutator) and the main routine are valid.*

4 Soundness

The structure of the soundness proof is as follows. We define a run-time checking execution semantics for the language of Figure 8. Execution gets stuck at null dereferences and assertion violations. We then define the notion of valid configuration. We show that valid programs do not get stuck by proving progress and preservation for valid configurations in valid programs. In the remainder of this section, we elaborate all the steps described above. For the full proof, we refer the reader to a technical report [7].

We start by defining an execution semantics for programs written in the language of Figure 8. More specifically, a configuration (H, S) consists of a heap H and a stack S. The former component is a partial function from object references to objects states. An object state is a partial function from field names to values. The stack consists of a list of activation records. Each activation record $(\Gamma, A, G, B, \overline{s})$ is a 5-tuple consisting of an environment Γ that maps variables to values, a set of accessible locations, an old heap G, an old access set B, and finally a sequence of statements. The old heap holds the value of the heap at the time the activation record was put onto the call stack, while the old access stores a copy of the callee's access set. A configuration can perform a step and get to a successor configuration as defined by the small-step relation $\rightarrow$. As an example, consider the definition of $\rightarrow$ for field update.

$$\frac{\begin{gathered}H, \Gamma, A \vdash e_1 \Downarrow v_1 \\ H, \Gamma, A \vdash e_2 \Downarrow v_2 \qquad (v_1, f) \in A \qquad H' = H[(v_1, f) \mapsto v_2]\end{gathered}}{(H, (\Gamma, A, G, B, e_1.f := e_2;\ \overline{s}) \cdot S) \rightarrow (H', (\Gamma, A, G, B, \overline{s}) \cdot S)}$$

In this definition, $H, \Gamma, A \vdash e_1 \Downarrow v_1$ denotes that the expression e_1 evaluates to value v_1. $H[(v_1, f) \mapsto v_2]$ denotes the update of the function H at location (v_1, f) with value v_2. Note that $\rightarrow$ defines a *run-time checking* semantics. For example, a field update is stuck if the location being assigned to is not in the activation record's access set. In general, $\rightarrow$ gets stuck at a null deference, precondition violation, postcondition violation, when a non-accessible location is read or written or when the condition of an assert statement evaluates to false.

$\rightarrow$ preserves certain well-formedness properties. In particular, it preserves the fact that (1) access sets of different activation records are disjoint and (2) that the access set of each activation record (except for the top of the stack) frames part of the heap with respect to the old heap. More specifically, for each activation record $(\Gamma_i, A_i, G_i, B_i, \overline{s_i})$, property (2) states that for all locations $o.f$ in A_i, the value of $o.f$ in the current heap H equals the value of $o.f$ in G_{i-1}. In other words, each location that is accessible to the callee but that is not required to be accessible by the caller cannot be changed during the callee's execution.

A configuration σ is *valid* if each activation record is valid. The top activation record $(\Gamma_1, A_1, G_1, B_1, \overline{s_1})$ is valid if $\mathfrak{I}, H, \Gamma_1, A \models \mathsf{vc}(\overline{s_1}, \psi_1)$, where ψ_1 is the postcondition of the method being executed. That is, the verification condition of the remaining statements satisfies the postcondition, when interpreting functions as defined in $\mathfrak{I}$, h as the heap H, a as the access set A and all variables by their value in Γ. Similarly, any other activation record $(\Gamma_i, A_i, G_i, B_i, \overline{s_i})$ is valid if $\mathfrak{I}, G_{i-1}, \Gamma_i, B_{i-1} \models \mathsf{vc}(\overline{s}, \psi_i)$.

Finally, we prove that for valid programs (i.e. for programs that successfully verify according to Definition 6) $\rightarrow$ preserves validity of configurations and that valid configurations are never stuck. In particular, we prove preservation for the return step by relying on the well-formedness properties described above. It follows that executions of valid programs do not violate assertions and never dereference null. Moreover, it is safe to *erase* the ghost state (e.g. access set per activation record) and the corresponding checks (e.g. that any location being assigned to is in the activation record's access set is accessible) in executions of valid programs.

Consistency For verification to be sound, the theory Σ_π must be consistent. To show consistency, it suffices to prove that $\mathfrak{I} \models \Sigma_\pi$ if π is a valid program. Since $\mathfrak{I} \models \Sigma_{prelude}$, it is sufficient to demonstrate that $\mathfrak{I}$ is a model for the axiomatization of each pure method. We prove the latter property by constructing a set of pure methods S, such that if a pure method p is in S, then all pure methods defined before p in the program text are also in S. We define Σ_S as the union of the axioms of all pure methods in S. We proceed by induction on the size of S. If S is empty, then trivially $\mathfrak{I} \models \Sigma_S$. If S is not empty, select the pure method p from S that appears last in the program text. It follows from the induction hypothesis that $\mathfrak{I} \models \Sigma_{p*}$. We have to show that $\mathfrak{I}$ is a model for each of p's axiom. The fact that $\mathfrak{I}$ models the implementation axiom follows from the fact that pure methods must terminate (i.e. a pure method can only call pure methods defined earlier in the program text) and the definition of $\mathfrak{I}$ for normal pure methods. The complete proof for all the axioms can be found in [7].

The main goal of our soundness proof is to show that the rules for framing are sound. We consider ensuring consistency of the logic in the presence of pure methods as an orthogonal issue. For that reason, we choose to ensure consistency in the proof by a very simple, but restrictive rule: a pure method p can only call pure methods defined before p in the program text. However, more flexible solutions exist [11, 13]. For example in our verifier prototype, we allow cycles in the call graph, provided the size of the precondition's required access set decreases along the call chain. Furthermore, a predicate may call any other predicate, provided the call occurs in a positive position.

5 Inheritance

Inheritance is a key component of the object-oriented programming paradigm that allows a class to be defined in terms of one or more existing classes. For example, the class *BackupCell* of Figure 11 extends its superclass *Cell* with a method *undo*. Dealing with inheritance in verification is challenging. In particular, for verification to be modular, the addition of a new subclass should not break or oblige us to reconsider the correctness of existing code. In this section, we informally describe how our approach can be extended to cope with Java-like inheritance in a modular way. Our approach for dealing with inheritance is based on earlier proposals by Leavens *et al.* [14], Parkinson *et al.* [15] and Jacobs *et al.* [12].

```
class BackupCell extends Cell {
  int backup;

  BackupCell()
    ensures valid() ∧ getX() = 0;
  { super(); }

  void setX(int v)
    requires valid();
    ensures valid();
    ensures getX() = v ∧ getBackup() = old(getX());
  { backup := super.getX(); super.setX(v); }

  void undo()
    requires valid();
    ensures valid() ∧ getX() = old(getBackup());
  { super.setX(backup); }

  predicate bool valid()
  { return acc(backup) ∗ super.valid(); }

  pure int getBackup()
    requires valid();
  { return backup; }
}
```

Fig. 11. A class *BackupCell* (similar to *Recell* from [15]) which extends *Cell* with *undo*.

Methods can both be statically and dynamically bound, depending on the method and the calling context. For example, *getX* is dynamically bound in the client code of Figure 7(b), while it is statically bound in the body of *setX* in Figure 11. To distinguish statically bound invocations of pure methods from dynamically bound ones, we introduce additional function symbols in the verification logic. That is, for a pure method p defined in a class C with parameters $x_1, \ldots, x_n$, the logic not only includes a function symbol $C.p$ but also a function $C.p_D$. The former function symbol is used for statically bound calls, while the latter is used for dynamically bound calls.

The relationship between $C.p$ and $C.p_D$ is encoded via a number of axioms. More specifically, $C.p$ equals $C.p_D$ whenever the dynamic type of the receiver (denoted as $\mathit{typeof}(\mathit{this})$) equals C.

$$\begin{gathered}\forall h, a, \mathit{this}, x_1, \ldots x_n \bullet \mathit{ok}(h, a) \wedge \mathit{typeof}(\mathit{this}) = C \Rightarrow \\ C.p(h, a, \mathit{this}, x_1, \ldots, x_n) = C.p_D(h, a, \mathit{this}, x_1, \ldots, x_n)\end{gathered}$$

Furthermore, whenever a method $D.p$ overrides $C.p$, we include the following axiom:

$$\begin{gathered}\forall h, a, \mathit{this}, x_1, \ldots x_n \bullet \mathit{ok}(h, a) \wedge \mathit{typeof}(\mathit{this}) <: D \Rightarrow \\ C.p_D(h, a, \mathit{this}, x_1, \ldots, x_n) = D.p_D(h, a, \mathit{this}, x_1, \ldots, x_n)\end{gathered}$$

That is, if the dynamic type of the receiver is a subtype (denoted as $<:$) of D, then dynamically bound invocations of both $C.p$ and $D.p$ yield the same result. For the footprint function of a predicate, we use a similar encoding.

Calls with receiver **this** are treated differently in code and in contracts. If a method invocation is statically bound, then invocations of pure methods with receiver **this** in the callee's contract are also considered to be statically bound; otherwise, such invocations are considered to be dynamically bound. Methods themselves are verified under the assumption they are called statically, i.e. calls with receiver **this** in the contract are statically bound. Doing so is sound, provided each subclass overrides each method. Indeed, if a method is called statically, then the caller and callee agree on the method contract. If a method is called dynamically, then the dynamic type of the receiver equals the static type, and therefore it follows that the static contract equals the dynamic contract.

To ensure the implementation of a subclass D does not break the contracts of a superclass C, we check that the contract of each method in C is satisfied by a method body that satisfies the contract of D. More specifically, for each method m in C, we check that a method body that calls $D.m$ satisfies the contract of $C.m$, assuming that the dynamic type of the receiver is D. The latter proof obligation ensures that no existing code is broken by the addition of the subclass C.

Note that *BackupCell* is just another client of *Cell* that is oblivious to *Cell*'s implementation. If we were to change *Cell*'s implementation (within the boundaries set by its method contracts), then the correctness of *BackupCell* would not be endangered.

6 Experience

To demonstrate the approach described in this paper is amenable to automatic, static verification, we implemented it in a verifier prototype. The prototype was used to verify several (variations of) programs used in related work.

The time taken to verify each program and a reference to the paper(s) containing the program is shown in Table 1. The experiments were executed on a desktop machine with a Pentium Core Duo 2.66 GHz processor and 4 GB of memory running Windows Vista. To discharge the verification conditions, we used the Z3 [10] theorem prover. The verifier itself and the programs shown in Table 1 can be downloaded from `http://www.cs.kuleuven.be/~jans/vericool2`.

program	time taken	source
Cell	0.1	[16, 17, 12]
ArrayList and Iterator	0.8	[2, 18]
LinkedList	43	[19, 2]
Resource Pool	2.1	[17]
Marriage	0.2	[20]
MasterClock	0.2	[21]
Subject-Observer	11	[1, 22]
Recell, TCell, DCell	0.5	[15]
Visitor (framing only)	127	[17]

Table 1. Table showing the time taken (in seconds) to verify each program.

To ensure a method's correctness proof does not depend on internal details of other modules, our verifier prototype makes a pure method's implementation axioms available only to other methods implemented in the same module.

Iterated Star In many programs, it is useful to specify that an assertion holds for a statically unknown number of objects. For example in the Subject-Observer pattern, the invariant of the subject typically states that all registered observers are valid. In our tool, there are two ways two write such invariants. First of all, one can define the invariant in terms of a recursive predicate. However, reasoning about recursive predicates in first-order provers is tricky, since this often involves proving inductive lemmas. To avoid reasoning about recursive predicates, our tool provides another way of quantifying over an unknown number of objects, namely iterated star. An iterated star assertion has the form $(\forall^* x \in (min : max) \bullet \phi)$, where *min* and *max* are integer expressions. Informally, the latter assertion states that ϕ holds for all integers between *min* (inclusive) and *max* (exclusive) and that for any two different integers in that range, the locations required to be accessible by ϕ are disjoint. For example, the invariant of the subject can be written as follows. Note that *obs* is a field of type $List < Observer >$.

```
predicate bool $subobs()$ {
  return $\mathbf{acc}(value) * \mathbf{acc}(obs) * obs \neq \mathbf{null} \wedge obs.valid() *$
  $(\forall^* i \in (0 : obs.size()) \bullet$
    $obs.get(i) \neq \mathbf{null} \wedge obs.get(i).valid() \wedge$
    $obs.get(i).getSubject() = \mathbf{this} \wedge obs.get(i).upToDate());$ }
```

We translate iterated star as follows (i and j are fresh variables).

$$(\forall x \bullet \mathsf{Tr}(min) \leq x < \mathsf{Tr}(max) \Rightarrow \mathsf{Tr}(\phi)) \wedge$$
$$(\forall i, j \bullet \mathsf{Tr}(min) \leq i < \mathsf{Tr}(max) \wedge \mathsf{Tr}(min) \leq j < \mathsf{Tr}(max) \wedge i \neq j \Rightarrow$$
$$\mathsf{R}(\phi[i/x]) \cap \mathsf{R}(\phi[j/x]) = \emptyset)$$

The first quantification states that ϕ holds for all integers between min and max, while the second one states that the required access set is disjoint at different indices. An iterated star is well-defined only if the bounds are well-defined and the assertion is well-defined for all integers within those bounds. That is, the definedness of an iterated star is as follows.

$$\mathsf{Df}(min) \wedge \mathsf{Df}(max) \wedge (\forall x \bullet \mathsf{Tr}(min) \leq x < \mathsf{Tr}(max) \Rightarrow \mathsf{Df}(\phi))$$

What is the required access set of an iterated star? Informally, the required access is the union of the required access sets of ϕ for all indices in the range: $\bigcup_{\mathsf{Tr}(min) \leq x < \mathsf{Tr}(max)} \mathsf{R}(\phi)$. However, $\bigcup$ is not a first-order concept. Therefore, we encode the required access set of an iterated star as follows (inspired by [23]). For each iterated star in the program text, we generate a function in the verification logic $union_i$ (where i is unique for each iterated star) with sort $heap \times set \times int \times int \rightarrow set$. This function represents the required access set of the corresponding iterated star. Several axioms describe the behavior of $union_i$. For example, we add an axiom that states a set is disjoint from a union only if it is disjoint from all the elements.

$$\forall h, a, min, max, s \bullet s \cap union_i(h, a, min, max) = \emptyset \Leftrightarrow$$
$$(\forall x \bullet \mathsf{Tr}(min) \leq x < \mathsf{Tr}(max) \Rightarrow s \cap \mathsf{R}(\phi) = \emptyset)$$

Whenever two different iterated stars are sufficiently similar, we generate only one union function instead of two. Two iterated stars are sufficiently similar if they differ only in the name of the quantified variable or in their range. Such similar iterated stars typically occur in loop invariants and postconditions.

Partial permissions

In this paper, we do not distinguish full access permissions (permission to read and write a location) from partial access permissions (permission to read). That is, a method either has permission to both read and write a location or it cannot access the location at all. Therefore, even if a mutator only reads an existing location, it still has to demand full access to that location in its precondition. This problem can be solved in many ways. For instance, Boyland [24] proposes using fractional permissions. We could extend our solution with support for fractions by tracking an access map, which maps each location to a fraction, instead of an access set.

In our implementation, we use a different solution. A mutator should indicate it only reads a location $o.f$ by ensuring in its postcondition that $o.f$'s value is not modified. However, a mutator's precondition can include predicates that its implementation relies on to call pure methods. In other words, the mutator might require a predicate to be true only to read locations protected by the predicate. Since the predicate's body may not be visible to the mutator, the mutator's postcondition may not be able to enumerate all those locations to ensure their value did not change. Therefore, our implementation includes a special assertion: **untouched**(ϕ). The assertion states that (1) all locations in ϕ's required access set have the same value in the old and the new heap, (2) those locations are still accessible in the new state and (3) the swinging pivot property holds for $\mathsf{R}(\phi)$.

```
class ArrayList {
  int n;  Object[] items;

  ArrayList()
    ensures valid() ∧ size() = 0;

  void add(Object o)
    requires valid();
    ensures valid();
    ensures size() = old(size() + 1);
    ensures (∀*i ∈ (0 : size() − 1)•
      get(i) = old(get(i)));
    ensures get(size() − 1) = o;

  predicate bool valid()
  { return acc(n) ∗ acc(items)∗
    items ≠ null ∗ acc_Elems(items)∗
    0 ≤ n ≤ items.length; }

  pure int size()
    requires valid();
  { return n; }

  pure Object get(int index)
    requires valid();
    requires 0 ≤ index < size();
  { return items[index]; }
}
```

```
class Iterator {
  ArrayList list;  int index;

  Iterator(List l)
    requires l ≠ null ∧ l.valid();
    ensures valid() ∧ getList() = l;
    ensures untouched(getList().valid());

  Object next()
    requires valid() ∧ hasNext();
    ensures valid();
    ensures getList() = old(getList());
    ensures untouched(getList().valid());

  predicate bool valid()
  { return acc(list) ∗ acc(index)∗
    list ≠ null ∧ list.valid()∗
    0 ≤ index ≤ list.size(); }

  pure bool hasNext()
    requires valid();
  { return index < list.size(); }

  pure bool getList()
    requires valid();
  { return list; }
}
```

Fig. 12. The iterator design pattern.

As an example, consider the classes *ArrayList* and *Iterator* from Figure 12. The last postcondition of the method *next* allows the verifier to deduce that other iterators of the same list remain valid. The conjunct $\mathbf{acc}_{Elems}(items)$ in *ArrayList*'s invariant is a special access assertion that gives permission to access the elements of the array. Also, note that it is ok for the invariant to read *items.length* without demanding access since *length* is immutable.

7 Related Work

The dynamic frames approach [1–6] solves the frame problem by explicitly annotating methods with effect annotations. More specifically, the contract of a mutator consists of a modifies clause and a "swinging pivot postcondition", while a pure method's contract includes a reads clause. The expressiveness of the dynamic frames approach stems from the fact that these effect annotations can mention arbitrary sets of memory locations. To support data abstraction, these location sets may be specified in terms of dynamic frames, i.e. pure methods or ghost fields that denote sets of locations. As an example, consider the dynamic frames version of the class *Cell* from Figure 7(a) (method *swap* not included) shown in Figure 13. *setX*'s contract includes a modifies clause indicating that

```
class Cell {
  int x;

  Cell()
    modifies ∅;
    ensures valid() ∧ getX() = 0;
    ensures fresh(footprint());
  { x := 0; }

  void setX(int v)
    requires valid();
    modifies footprint();
    ensures valid() ∧ getX() = v;
    ensures fresh(footprint()
      \old(footprint()));
  { x := v; }

  pure bool valid()
    reads footprint();
  { return true; }

  pure set footprint()
    reads footprint();
  { return { (this, x) }; }

  pure int getX()
    requires valid();
    reads footprint();
  { return x; }
}
```

Fig. 13. The class *Cell* with traditional dynamic frames annotations.

all locations in the dynamic frame *footprint* can potentially be changed by the method. In addition, *setX*'s last postcondition encodes the swinging pivot property. The contract of each pure method includes a reads clause indicating that its return value only depends on locations in *footprint*(). All the latter effect annotations (indicated with the grey background) need to be provided by the developer, and must be checked explicitly by the verifier. In our approach on the other hand, none of the annotations in grey need to be provided or checked explicitly (they are free postconditions!). Instead, we only check at each field access that the corresponding location is accessible, which allows us to deduce an upper bound on the set of readable and writable locations. Since access assertions can typically be piggy-backed onto invariants, as shown in the predicate *valid* of class *ArrayList* of Figure 12, contracts do not need to include additional effect annotations. Moreover, as callers typically already have to establish a callee's invariant and the invariant is opaque to the caller, checking the access assertions inside the callee's precondition incurs no additional cost.

Our approach was heavily inspired by separation logic [16, 15, 25]. In particular, the access assertion $\mathbf{acc}(e.f)$ is similar to separation logic points-to predicate $e.f \mapsto \_$ and Parkinson and Bierman's abstract predicates inspired our predicate pure methods. To the best of our knowledge, this is the first approach based on verification condition generation and automatic, first-order theorem proving that encodes separation logic's idea of deducing frame information from preconditions. One difference between separation logic and implicit dynamic frames is that we allow using heap-dependent expressions, in particular field reads and pure method invocations, inside assertions. Distefano and Parkinson [17] recently implemented a verifier for Java based on separation logic, called jStar. jStar relies on symbolic execution, while we use the more traditional combination of verification condition generation and automated theorem proving. The access set used in our verification conditions resembles the coloring of objects used in SLICK [26] for runtime checking of separation logic assertions.

In [27], the authors propose using data groups to specify side-effects. To ensure soundness, their approach imposes two methodological restrictions: the pivot uniqueness and owner exclusion restriction. Our approach imposes no such restrictions, and as a consequence it can handle programs that [27] cannot. For example, the former restriction rules out sharing of representation objects, as is the case in the iterator pattern.

In the universe type system [28] and the Boogie methodology [29], abstractions (pure methods, invariants or model fields) can depend on the fields of owned objects and the fields of peers (i.e. objects with the same owner as the receiver), provided the abstraction is visible to the peer. For example, the method *hasNext* of an iterator would have to be visible to the list class. Our approach has no such restriction.

The use of pure methods in specifications has been discussed extensively in the literature [11–13]. In particular, encoding pure methods as functions in the logic is a standard technique in verification. To the best of our knowledge, this is the first approach that derives an upper bound on the set of readable locations from preconditions of pure methods. Some authors propose broadening the range of admissible pure methods by allowing certain side-effects. We believe our approach can be extended to support such weakly pure methods.

Verification of Java-programs with JML-like [30] annotations has received considerable attention in the research community [30–32]. To the best of our knowledge, all the JML tools rely on explicit effect annotations for framing. We believe those tools might benefit from our approach to reduce the number of effect annotations.

Zee *et al.* [19] focus on verification of linked data structures. Their technique for dealing with such data structures inspired our specification of linked list. In particular, they use a ghost field to represent the set of all nodes in a list and rely on quantification over that set in the invariant to appropriately constrain the values and next pointers of the list.

A preliminary version of this work was presented at the 2008 FTFJP workshop [33]. This preliminary version already sparked the interest of other authors [34]. In particular, Leino and Müller combine implicit dynamic frames with fractional permissions and concurrency. However, they encode accessibility differently and do not show how to deal with data abstraction or inheritance in their encoding. Moreover, they provide no formal soundness proof.

8 Conclusion

In this paper, we improve upon the classical dynamic frames approach in two ways: (1) method contracts are more concise and (2) fewer proof obligations must be discharged by the verifier. We have proven soundness, implemented the approach in a verifier prototype and demonstrated its expressiveness by verifying several challenging examples from related work.

In the future, we plan to extend our approach to concurrent programs.

Acknowledgments

Jan Smans is a research assistant of the Fund for Scientific Research - Flanders (FWO). Bart Jacobs is a postdoctoral fellow of the Fund for Scientific Research - Flanders (FWO). This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy.

References

1. Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, 2008.
2. Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, 2006.
3. Anindya Banerjee, Mike Barnett, and David A. Naumann. Boogie meets regions: a verification experience report. In *VSTTE*, 2008.
4. K. Rustan M. Leino. Specification and verification of object-oriented software. In *Marktoberdorf International Summer School*, 2008.
5. Bernd Schoeller. *Making Classes Provable through Contracts, Models and Frames*. PhD thesis, Departement Informatik ETH Zurich, 2007.
6. Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for java-like programs based on dynamic frames. In *FASE*, 2008.
7. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic (soundness proof). Technical Report CW542, Katholieke Universiteit Leuven, 2009.
8. Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS*, 2007.
9. Christian Haack and Clement Hurlin. Separation logic contracts for a java-like language with fork/join. In *AMAST*, 2008.

10. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.
11. Arsenii Rudich, Ádám Darvas, and Peter Müller. Checking well-formedness of pure-method specifications. In *FM*, 2008.
12. Bart Jacobs and Frank Piessens. Inspector methods for state abstraction. *Journal of Object Technology*, 6(5), 2007.
13. K. Rustan M. Leino and Ronald Middelkoop. Proving consistency of pure methods and model fields. In *FASE*, 2009.
14. Gary T. Leavens. JML's rich, inherited specifications for behavioral subtypes. In *ICFEM*, 2006.
15. Matthew Parkinson and Gavin Bierman. Separation logic, abstraction and inheritance. In *POPL*, 2008.
16. Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, 2005.
17. Dino Distefano and Matthew Parkinson. jStar: Towards practical verification for java. In *OOPSLA*, 2008.
18. Matthew Parkinson. *Local Reasoning for Java.* PhD thesis, University of Cambridge, 2005.
19. Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In *PLDI*, 2008.
20. M. Leino K. Rustan and Peter Müller. Object invariants in dynamic contexts. In *ECOOP*, 2004.
21. Mike Barnett and David A Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, 2004.
22. Matthew Parkinson. Class invariants: The end of the road? In *IWACO*, 2007.
23. K. Rustan M. Leino and Rosemary Monahan. Automatic verification of textbook programs that use comprehensions. In *FTFJP*, 2007.
24. John Boyland. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium*, 2003.
25. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
26. Huu Hai Nguyen, Viktor Kuncak, and Wei-Ngan Chin. Runtime checking for separation logic. In *VMCAI*, 2008.
27. K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *PLDI*, 2002.
28. Peter Müller. *Modular Specification and Verification of Object-Oriented Programs.* PhD thesis, FernUniversität Hagen, 2001.
29. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6), 2003.
30. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. 1999.
31. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.
32. Cormac Flanagan, K. Rustan, M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI*, 2002.
33. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. In *FTFJP*, 2008.
34. K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In *ESOP*, 2009.