Automatic verification of Java programs with dynamic frames

Jan Smans, Bart Jacobs, Frank Piessens, Wolfram Schulte

Under consideration for publication in Formal Aspects of Computing

Automatic Verification of Java Programs with Dynamic Frames

Jan Smans[1], Bart Jacobs[1], Frank Piessens[1] and Wolfram Schulte[2]

[1] Katholieke Universiteit Leuven, Belgium
[2] Microsoft Research Redmond, USA

Abstract. Framing in the presence of data abstraction is a challenging and important problem in the verification of object-oriented programs [LLM07]. The dynamic frames approach is a promising solution to this problem. However, the approach is formalized in the context of an idealized logical framework. In particular, it is not clear the solution is suitable for use within a program verifier for a Java-like language based on verification condition generation and automated, first-order theorem proving.

In this paper, we demonstrate that the dynamic frames approach can be integrated into an automatic verifier based on verification condition generation and automated theorem proving. The approach has been proven sound and has been implemented in a verifier prototype. The prototype has been used to prove correctness of several programming patterns considered challenging in related work.

Keywords: program verification, dynamic frames, frame problem, data abstraction

1. Introduction

How does one specify which memory locations a method can touch? This problem is called the *frame problem* [BMR95]. As an example, consider the class *Cell* from Figure 1(a). This class provides a constructor for creating new instances and a mutator for modifying their state. The client code in Figure 1(b) uses these methods to create and modify two *Cell* objects. At the end of the code snippet, an assert statement checks that $c_1.x$ holds 5. Can we prove based on *Cell*'s method contracts that this assertion never fails? The answer is no. *setX*'s contract allows the method to change the program state arbitrarily, as long as it ensures that $c_1.x$ equals v on exit. In particular, the contract does not prevent $c_2.setX(10)$; from modifying $c_1.x$.

Modifies clauses are a standard technique in verification to solve the frame problem. A modifies clause traditionally consists of a comma-separated list of memory locations. A method satisfies its modifies clause if all allocated locations except those listed in the modifies clause retain their value. For example, consider the new version of *Cell* from Figure 2. *setX*'s modifies clause only lists the location **this**.x which indicates that the method can only change the value of that particular location. A modifies clause does not prevent a method from modifying fields of objects allocated during execution of the method itself. For example,

Correspondence and offprint requests to: Jan Smans, Departement of Computer Science, Celestijnenlaan 200A, 3001 Heverlee, Belgium. e-mail: jan.smans@cs.kuleuven.be

```
class Cell {
  int x;

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

  void setX(int v)
    ensures x = v;
  { 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.

```
class Cell {
  int x;

  Cell()
    modifies nothing;
    ensures x = 0;
  { x := 0; }

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

Fig. 2. A new version of the *Cell* with modifies annotations.

although the modifies clause of the constructor contains no locations, it can modify the fields of the object created by the constructor[1].

The new contracts for *Cell* shown in Figure 2 suffice to prove that the assert statement of Figure 1(b) never fails. Informally, the reasoning is as follows. At program location A, the postcondition of $c_1.setX(5)$; holds: $c_1.x = v$. Since c_1 is allocated and c_2's constructor does not modify fields of allocated objects, $c_1.x$ still equals 5 after the constructor. Moreover, c_2 is different from c_1, since c_2 is a newly allocated object. It follows from *setX*'s modifies annotation that $c_2.setX(10)$; only modifies $c_2.x$ and does not affect $c_1.x$. We may conclude that the condition of the assert statement, $c_1.x = 5$, always evaluates to true.

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* and its contracts were not written with data abstraction in mind. In particular, client code must directly access the internal field x to query a *Cell*'s internal state. Moreover, *Cell*'s method contracts are not implementation-independent as they mention x. To solve the former issue, programmers typically restrict the visibility of internal fields and add getters to their classes. For example, Figure 3(a) extends *Cell* with a getter *getX*. The client code of Figure 3(b) can now use the getter instead of the internal field x. Note that *getX* has a **pure** modifier which indicates the method does not have side-effects and that it can be used within method contracts. In the remainder of this paper, we use the terms getter and pure method interchangeably.

To complete the decoupling between *Cell* and its clients, we must remove all implementation-specific elements from the contract. That is, instead of referring to the internal field x, we should specify the behavior in terms of publicly visible parts of *Cell*'s interface. For the postconditions, this can be accomplished by

[1] This description of object creation is slightly simplified. In our verifier prototype, object creation is modeled by first allocating new memory locations and afterwards calling the constructor. Each constructor in a class C has a default modifies clause that allows it to modify fields of **this** in C and superclasses of C.

```
class Cell {
  int x;

  Cell()
    modifies nothing;
    ensures getX() = 0;
  { x := 0; }

  void setX(int v)
    modifies ?;
    ensures getX() = v;
  { x := v; }

  pure int getX()
  { return x; }
}
```

(a)

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

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

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

(b)

Fig. 3. A class *Cell* with a getter and client code which uses the getter.

specifying the behavior in terms of *getX*. For example, *setX*'s postcondition guarantees that *getX*() returns v on exit. However, how do we indicate what locations *setX* can modify without exposing the fact that *Cell* internally uses the field x? Moreover, client code does not know which memory locations the return value of *getX* depends on. How can we extend *Cell*'s interface to ensure that $c_2.setX(10)$; does not affect $c_1.getX()$'s return value in the code snippet of Figure 3(b)? This problem is called the *frame problem in the presence of data abstraction*. Leavens *et al.* [LLM07] consider this problem to be one of the most important challenges in verification of sequential object-oriented programs.

A naive solution to this problem would be to adapt the semantics of modifies clauses: instead of listing the memory locations that can be modified, they could list the set of getters whose return value may be affected by the mutator. For example, *setX*'s modifies clause would be **this**.*getX*(). Although the latter approach suffices to prove the assert statement of Figure 3(b) never fails, it is not a modular solution since the addition of a new class may break *setX*'s contract. For example, suppose we extend the program of Figure 3 with a new class *CellWrapper*. Suppose *CellWrapper* has a field referring to a *Cell* object and a pure method *getValue* which returns the value of that cell. This extension breaks *setX*'s contract as its modifies clause should now additionally mention *getValue*. Clearly, this approach does not scale to larger programs.

A promising solution to the frame problem was proposed by Yannis Kassios in 2006. More specifically, Kassios proposes adding sets of locations to the specification language. The footprint of a method, i.e. the locations read or written during the method's execution, can then be specified in terms of such sets. To preserve information hiding, these sets of locations can involve dynamic frames, specification variables that abstract over a set of locations. Kassios' solution is formalized in the context of an idealized logical framework. In particular, it is not clear the solution is suitable for use within a program verifier for a Java-like language based on verification condition generation and automated, first-order theorem proving [Kas06b, Kas06a]. Several researchers [JP07, LS07, DL07] called for the integration of the dynamic frames approach into such a program verifier.

In this paper, we demonstrate how the dynamic frames approach can be integrated into an automatic verifier for a Java-like language based on verification condition generation and automated, first-order theorem proving. In particular, we implemented our approach in a tool and used it to verify a number of challenging patterns, including iterator and subject-observer. In addition, we have proven its soundness. This paper is based on and extends a paper published at the Conference on Fundamental Approaches to Software Engineering 2008 [SJPS08]. In particular, this paper extends its predecessor by formalizing the verification condition generation and by proving its soundness.

The remainder of this paper is structured as follows. Section 2 describes how the dynamic frames approach solves the frame problem in the presence of data abstraction. We illustrate this solution and its expressive power using various examples in Section 3. In Section 4, we show how the programs described in earlier sections can be automatically verified using an SMT solver. More specifically, we define the notion of valid

```
class Cell {
  int x;

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

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

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

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

Fig. 4. A class *Cell* with frame annotations written in terms of the dynamic frame *footprint*.

program. Informally, a program is valid if the verification condition of each method is a valid, first-order formula. We prove the soundness of the verification technique, i.e. that valid programs never dereference null and that assert statements never fail, in Section 5. We extend the approach described in the previous sections to inheritance, discuss experience with a prototype implementation, compare with related work and conclude in Sections 6, 7, 8 and 9.

2. Dynamic Frames

The dynamic frames approach solves the frame problem in the presence of data abstraction via dynamic frames. A *dynamic frame* is a special, specification-only pure method that returns a set of memory locations. As an example, consider the method *footprint* of Figure 4. The method's return type, **set**, declares that *footprint* returns a set of memory locations. A memory location is an (object reference, field name) pair. *footprint*'s implementation returns the singleton set containing the memory location **this**.x. In this paper, we denote sets of memory locations using the standard mathematical notation.

How do we indicate what locations *setX* can modify without exposing the fact that *Cell* internally uses the field x? We can do so by adapting the semantics of modifies clauses as follows. A modifies clause no longer lists a sequence of memory locations, but instead consists of a single expression of type **set**. For example, *setX*'s modifies clause states that the method can modify any location in the set returned by the dynamic frame *footprint*(). Note that *setX*'s contract is now implementation-independent, as its modifies clause only mentions publicly visible elements of *Cell*'s interface.

To indicate which locations influence the return values of pure methods, we enrich their contracts with reads annotations. Similarly to a modifies annotation, a reads annotation consists of an expression of type **set**. For example, the reads clause for *getX* in Figure 4 declares that *getX*'s return value depends only on locations in the set returned by *footprint*(). The method *footprint* itself specifies that it is *self-framing*: if *footprint* reads a location $o.f$, then $o.f$ is an element of its result. Note that **reads** ∅; would also be a valid reads clause for *footprint*. However, the latter annotation would expose and unnecessarily constrain *Cell*'s implementation. In particular, it would expose the fact that *Cell* stores its value directly in one of its own field and precludes reimplementing *Cell* as an aggregate object.

Both *Cell*'s constructor and *setX* have an extra postcondition which states that (part of) *footprint*()

is fresh. A set of memory locations s is fresh if each location in s is fresh. A location $o.f$ is *fresh* with respect to a method invocation $e_0.m(e_1, \ldots, e_n)$; if o was allocated during m's execution. In our running example, *setX*'s postcondition ensures that *setX* only adds fresh locations to *footprint*. This property is called the swinging-pivot property [LN02] and it is crucial for ensuring that footprints remain disjoint as we will explain below. Note that instead of the swinging pivot property, we could have used the following, stronger postcondition for *setX*: $footprint() = \mathbf{old}(footprint())$. We did not do so as the latter postcondition precludes certain different, valid implementations of *Cell*.

Given the semantics of reads and writes clauses outlined above, we can now prove that the assertion at the end of Figure 3(b) never fails. Informally, the reasoning is as follows. At program location A, $c_1.setX(5)$;'s postcondition holds: $c_1.getX() = 5$. It follows from the read annotations on *getX* and *footprint* that the return values of $c_1.getX()$ and $c_1.footprint()$ depend only on locations in $c_1.footprint()$. A general property of dynamic frames is that they contain only allocated locations. Hence, $c_1.footprint()$ does not contain any non-allocated locations. c_2's constructor does not modify any allocated location. Hence, the return values of $c_1.getX()$ and $c_1.footprint()$ are not affected by the constructor. Moreover, the constructor's postcondition guarantees that $c_2.footprint()$ contains only fresh locations and therefore $c_1.footprint()$ and $c_2.footprint()$ are disjoint. Since $c_2.setX(10)$; only assigns to locations in $c_2.footprint()$ and because of the disjointness of the footprints, $c_1.getX()$ still returns 5 after $c_2.setX(10)$;. We may conclude that the condition of the assert statement, $c_1.getX() = 5$, always evaluates to true. Moreover, it follows from *setX*'s swinging pivot postcondition that $c_1.footprint()$ and $c_2.footprint()$ are still disjoint.

Note that the proof outlined above does not depend on internal details of the class *Cell*. Therefore, changing *Cell*'s implementation (within the boundaries set by its contracts) preserves the correctness of the client code in Figure 3(b).

3. Examples

In this section, we demonstrate how various object-oriented programming and specification patterns can be handled using dynamic frames. More specifically, we focus on object invariants (3.1), aggregate objects (3.2) and sharing (3.3). It is important to note that supporting these patterns does *not* require any special methodological machinery or reasoning rules.

3.1. Invariants

An invariant describes what it means for an object to be in a consistent state. For example, consider the class *ArrayList* from Figure 5. An *ArrayList* object is consistent if the internal array, *items*, is non-null and *size* is non-negative and less than or equal to the array's length.

In the dynamic frames approach, object invariants are encoded as Boolean pure methods. For example, the pure method *valid* of Figure 5 embodies *ArrayList*'s invariant. The approach does not impose any built-in rules which guide when invariants hold. Instead, if a developer wants to indicate a method m requires or ensures an object is consistent, he or she must explicitly state the invariant method returns true in m's method contract. For instance, the method *add* requires that the invariant holds on entry and ensures it holds again on exit. *add*'s method contract specifies this property by stating in its pre- and postcondition that *valid*() returns true.

The traditional approach for dealing with object invariants[2] is unsound in the presence of callbacks. The dynamic frames approach is not vulnerable to callbacks, since it never assumes more than it proves. That is, if a callee assumes that an object is consistent, then the caller must explicitly prove this. For example, it is safe for *add* to assume the invariant holds, since its callers must prove its precondition (which includes the invariant).

footprint() is well-defined only if the invariant holds. Therefore, we must make *valid*'s reads clause conditional: if *valid*() returns true, then its return value depends only locations in *footprint*(); otherwise, *valid*() can potentially depend on the entire heap (denoted as $*$). Since client code typically only sees valid

[2] In the traditional approach, callers of T's methods do not need to be concerned with establishing the implicit precondition associated with the invariant. For the invariant of a class T to hold at entries to its public methods, it is sufficient to restrict modifications of the invariant to methods of T and for each method in T to establish the invariant as a postcondition. [BDF+04]

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

  ArrayList()
    modifies ∅;
    ensures valid() ∧ size() = 0 ∧ fresh(footprint());
  { items := new Object[5]; size := 0; }

  void add(Object o)
    requires valid();
    modifies footprint();
    ensures valid() ∧ size() = old(size()) + 1;
    ensures get(size() − 1) = o;
    ensures ∀i ∈ (0 : size() − 1) • get(i) = old(get(i));
    ensures fresh(footprint() \ old(footprint()));
  { ... }

  pure bool valid()
    reads valid() ? footprint() : ∗;
  { return items ≠ null ∧ 0 ≤ size ≤ items.length; }

  pure set footprint()
    requires valid();
    reads footprint();
  { return { items, size, items.elems }; }

  pure int size()
    requires valid();
    reads footprint();
  { return size; }

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

Fig. 5. The class *ArrayList*.

Cell objects and does not care whether *Cell* objects *remain* invalid, it suffices to frame the invariant only if it holds. An alternative choice would be to remove *footprint*'s precondition and to make *valid*'s reads clause unconditional. However, this choice means we have to make sure *footprint*'s implementation can deal with invalid states, which typically leads to duplicating parts of the invariant in the footprint method. For example, the implementation of *footprint* would have to be changed as follows:

$$\textbf{return } \{ \mathit{items}, \mathit{size} \} \cup (\mathit{items} \neq \mathit{null} \ ? \ \{ \mathit{items.elems} \} : \emptyset);$$

The condition $\mathit{items} \neq \mathit{null}$ then appears both in *valid* and *footprint*.

Note that *ArrayList*'s footprint includes *items.elems*. We assume each array has an *elems* field which holds a mathematical list containing the elements of the array. We model array updates by placing a new list value in the *elems* field. Updating an array corresponds to placing a new list value in the elems field. Including this field in a reads or modifies clause, gives a method the right to read, respectively modify the array's elements. For example, the method *get* is allowed to mention *items*[*index*], since the footprint includes *items.elems*. Note that the invariant can read *items.length* without requiring access to this field in

```
class Stack {
  ArrayList contents;

  Stack()
    modifies ∅;
    ensures valid() ∧ size() = 0;
  { contents := new ArrayList(); }

  void push(Object o)
    requires valid();
    modifies footprint();
    ensures valid() ∧ size() = old(size() + 1);
    ensures fresh(footprint() \ old(footprint()));
  { contents.add(o); }

  pure bool valid()
    reads valid() ? footprint() : ∗;
  { return contents ≠ null ∧
      contents.valid() ∧ {contents} ∩ contents.footprint() = ∅; }

  pure set footprint()
    requires valid();
    reads footprint();
  { return { contents } ∪ contents.footprint(); }

  pure int size()
    requires valid();
    reads footprint();
  { return contents.size(); }
}
```

Fig. 6. The class *Stack* implemented on top of *ArrayList*.

its reads clause, since *length* is immutable. Finally, the variable i in a quantifier $\forall i \in (a : b) \bullet \ldots$ ranges from a (inclusive) to b (exclusive).

3.2. Aggregate Objects

A software program typically consists of several modules, where each module is implemented in terms of other modules defined in lower layers. For example, a typical way to implement a stack is to use a collection, such as an *ArrayList*, which holds the stack's elements. Such objects that internally use other helper objects are sometimes called *aggregate objects*. Typically, the consistency of an aggregate object implies consistency of all its helper objects and an aggregate object's footprint includes the footprints of all its helper objects.

As an example, consider the class *Stack* from Figure 6. This class implements stacks on top of the class *ArrayList*. The dynamic frame *footprint* includes the footprint of the internal *ArrayList* object pointed to by the *contents* field. *Stack*'s invariant states that *contents* is not null, that *contents*'s invariant holds and that *contents*'s footprint does not include the location **this**.*contents* itself. The latter disjointness restriction is crucial for two reasons. First of all, it ensures that updates to *contents* do not affect the return values of pure methods of the *ArrayList* object. Secondly, the restriction guarantees that *contents*'s mutators cannot modify **this**.*contents*. For example, the call *contents.add*(*o*); in the method *push* cannot set *contents* to null (and thus violate the invariant), since *contents.footprint*() (i.e. the set of locations modifiable by *add*) does not include **this**.*contents*. The swinging pivot property, the last postcondition of *ArrayList.add*, is key in preserving disjointness between *contents.footprint*() and { *contents* }.

Note that *Stack*'s interface is completely implementation-independent. In particular, the interface does

not expose that the class is internally implemented in terms of an *ArrayList* object. Therefore, if we were to change *Stack*'s implementation, for example by using a linked list instead of an array list, client code would not be affected. That is, if one makes internal changes to *Stack*, one has to reverify that the new implementation satisfies the specification. However, one does not have to reverify any client code. In other words, implementation-independent contracts ensure modularity. Moreover, note that *Stack* itself does not depend on internal details of its helper *ArrayList* object. We can therefore freely change the inner workings of the class *ArrayList* without having to worry about *Stack* or other potential clients.

Our approach does not impose any (built-in) aliasing restrictions. In particular, it does not forbid an aggregate object from leaking references to its internal helper objects. For example, a *Stack* is allowed to pass out references to its internal *ArrayList* object to client code. However, client code will not be able to establish disjointness between the stack's footprint and the footprint of the *ArrayList* object. As a consequence, when client code updates the helper object, it loses all information about the aggregate object. In particular, the client may not assume that the pure methods of the *Stack* object return the same value before and after the update (because their frame axioms cannot be applied, see Section 4). Therefore, clients cannot falsely assume that the state of an aggregate object is not affected when one of the helper objects is modified.

A dynamic frame is called *dynamic* for two reasons. First of all, the return value of a dynamic frame can change dynamically over time. For example, the set of locations returned by *footprint*() is different before and after invoking the method *switch* shown below. Exchanging an internal helper object, as in the method *switch*, is sometimes called *ownership transfer*, since control of the helper object transfers from one aggregate object to another. No additional methodological machinery is needed to perform an ownership transfer.

```
void switch(Stack other)
  requires other ≠ null;
  requires valid() ∧ other.valid();
  requires footprint() ∩ other.footprint() = ∅;
  modifies footprint() ∪ other.footprint();
  ensures valid() ∧ other.valid();
  ensures footprint() ∩ other.footprint() = ∅;
  ensures fresh((footprint() ∪ other.footprint())
    \ old(footprint() ∪ other.footprint()));
  ensures size() = old(other.size());
  ensures other.size() = old(size());
{
  ArrayList tmp := contents;
  contents := other.contents;
  other.contents := tmp;
}
```

The second reason why dynamic frames are dynamic is that they can be overridden in subclasses as to include additional locations (see Section 6).

3.3. Sharing

Figure 7 shows an implementation of the iterator pattern. Reasoning about the iterator pattern is challenging because of *sharing*: when multiple iterators share the same list, the footprints of those iterators overlap. More specifically, the invariants of those iterators all require the list's invariant holds. Therefore, the reads clause of *Iterator.valid* must include the list's footprint.

Multiple iterators can iterate over the same list at the same time, because an iterator's *next* method only modifies its own footprint, but not the footprint of the shared list. For example, the code of Figure 8 successfully verifies. For our verification tool to be able to reason about loops, developers must provide loop invariants and loop modifies annotations. A loop invariant must hold on entry to the loop and must be preserved by each iteration of the loop. Similarly, a loop iteration may only modify a location $o.f$ if $o.f$ is in the loop's modifies clause. Old expressions **old**(e) appearing in the loop invariant refer to the value of the expression e on entry to the loop.

Modifying a list while iterators are iterating over it can give rise to unexpected exceptions. For example,

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

  Iterator(ArrayList l)
    requires l ≠ null ∧ l.valid();
    modifies ∅;
    ensures valid();
    ensures getList() = l;
    ensures fresh(footprint());
  { list := l; index := 0; }

  Object next()
    requires valid();
    requires hasNext();
    modifies footprint();
    ensures valid();
    ensures getList() =
      old(getList());
    ensures fresh(footprint() \
      old(footprint()));
  { return list.get(index++); }

  void reset()
    requires valid();
    modifies footprint();
    ensures valid();
    ensures getList() =
      old(getList());
    ensures fresh(footprint() \
      old(footprint()));
  { index := 0; }

  pure bool valid()
    reads valid() ?
      footprint() ∪ getList().footprint()
      : *;
  { return list ≠ null ∧ list.valid() ∧
      ({ list, index } ∩
        list.footprint() = ∅) ∧
      0 ≤ index ≤ list.size(); }

  pure set footprint()
  { return { list, index }; }

  pure ArrayList getList()
    requires valid();
    reads footprint();
  { return list; }

  pure bool hasNext()
    requires valid();
    reads footprint() ∪
      getList().footprint();
  { return index < list.size(); }
}
```

Fig. 7. The class *Iterator*.

removing an element from the list can give rise to an *ArrayOutOfBoundsException* when calling the iterator's *next* method. Java's implementation of *Iterator* guards against concurrent modifications by tracking and checking a version field. Our implementation does not need to track such a field to stay safe. Instead, when an *ArrayList* object is modified, all its iterators are automatically invalidated. Indeed, the method *Iterator.valid* reads the footprint of its list. Hence, any modification to the list causes all information about the invariant to be lost (since we cannot apply the frame axiom, see Section 4).

4. Verification

In this section, we show how one can verify whether programs, such as the ones shown above, satisfy their specification. More specifically, we define the notion of valid program. Informally, a program is valid if the verification conditions of all methods in the program are valid, first-order formulas. In Section 5, we will prove that executions of valid programs never dereference null and that assert statements never fail.

Note that we formalize verification only for a subset of Java. For example, this formalization does not include arrays, if-then-else statements, loops, and exceptions.

```
ArrayList l := new ArrayList();

Iterator i1 := new Iterator(l);
Iterator i2 := new Iterator(l);

while(i1.hasNext())
  invariant i1.valid() ∧ i1.getList() = l;
  invariant i2.valid() ∧ i2.getList() = l;
  invariant i1.footprint() ∩ i2.footprint() = ∅;
  invariant fresh((i1.footprint() ∪ i2.footprint())
      \old(i1.footprint() ∪ i2.footprint()));
  modifies i1.footprint() ∪ i2.footprint();
{
  Object o1 := i1.next();
  while(i2.hasNext())
    invariant i2.valid();
    invariant fresh(i2.footprint() \ old(i2.footprint()));
    invariant i2.getList() = l;
    modifies i2.footprint();
  {
    Object o2 := i2.next();
    ...
  }
  i2.reset();
}
```

Fig. 8. Client code for *Iterator*.

program	::=	$\overline{class}\ \overline{s}$
class	::=	**class** C { $\overline{field}$ $\overline{method}$ }
field	::=	$C\ f$;
method	::=	*mutator* | *pure*
mutator	::=	**void** $m(\overline{C\ x})$ **requires** e; **modifies** e; **ensures** e; { $\overline{s}$ }
pure	::=	**pure** $t\ p(\overline{C\ x})$ **requires** e; **reads** e; { **return** e; }
t	::=	C | **bool** | **set**
s	::=	$C\ x$; | $e.f := e$; | $x := e$; | $e.m(\overline{e})$; | $x :=$ **new** C; | **assert** e;
e	::=	*null* | x | $e.f$ | $e.p(\overline{e})$ | **old**(e) | **true** | $e = e$ | $e \wedge e$ | **fresh**(e) | $e\ ?\ e : e$ | $\emptyset$ | { $e.f$ } | $e \cup e$ | $e \setminus e$

Fig. 9. The syntax of a Java-like language with method contracts.

4.1. Language

We start by defining the following sets.

set	typical element	meaning
$\mathcal{C}$	C	class names
$\mathcal{F}$	f	field names
$\mathcal{M}$	m	mutator names
$\mathcal{P}$	p	pure method names
$\mathcal{X}$	x	variable names

All these sets are mutually disjoint. $\mathcal{X}$ includes the variable **this**. Programs have the following syntax. An overlined program element indicates repetition.

A program consists of a number of classes and a main routine $\overline{s}$. Each class declares a number of fields and methods. We distinguish two kinds of methods: mutators and pure methods. Each method has a corresponding contract. The contract of a pure method consists of a precondition and a reads clause, while a mutator's contract contains a precondition, a modifies clause and a postcondition. The body of a mutator consists of a sequence of statements. A statement is either a local variable declaration, a field update, a variable update, a mutator invocation, an object creation or an assert statement. An expression is either the constant **null**, a variable read, a field read, a pure method invocation, an old expression, the constant **true**, an equality between expressions, a conjunction, a fresh expression, a conditional expression, the empty set, a singleton, a union, or a subtraction.

In this paper, we consider only well-formed programs.

Definition 1. A program π is *well-formed* (denoted $\mathsf{wf}(\pi)$) if all of the following hold:

- **fresh** and **old** expressions only appear in postconditions.
- Method parameters are not assigned to.
- The program is well-typed. We do not formalize the type system of our language in this paper as the formalization would be similar to [DE97, BPP03, IPW99].
- The body and precondition of a pure method only call pure methods defined earlier in the program text. Note that the latter restriction does not apply to the reads clause.

The last well-formedness criterion is used in the soundness proof (Section 5) to ensure consistency of the verification logic. More liberal schemes for ensuring consistency can be found in the literature (e.g. [DL07, RADM08]). The well-typedness criterion is used to ensure the program does not get stuck due to Field- and MethodNotFound errors.

4.2. Verification Logic

We target a multi-sorted, first-order logic with equality. That is, a formula ψ is either *true*, *false*, an equality between terms, a conjunction, a disjunction, a negation or a quantification. A term τ is a variable or a function application.

$$\begin{array}{lcl} \psi & ::= & \mathit{true} \mid \mathit{false} \mid \tau = \tau \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg\psi \mid \forall \overline{y} \bullet \psi \\ \tau & ::= & y \mid g(\overline{\tau}) \end{array}$$

Throughout this paper, we will use the following shorthands. First of all, $\psi_1 \Rightarrow \psi_2$ is a shorthand for $\neg\psi_1 \vee \psi_2$, $\psi_1 \Leftrightarrow \psi_2$ is a shorthand for $\psi_1 \Rightarrow \psi_2 \wedge \psi_2 \Rightarrow \psi_1$, and $\tau_1 \neq \tau_2$ is a shorthand for $\neg(\tau_1 = \tau_2)$. Secondly, an application of a function g with no parameters will be written g instead of $g()$. In this paper, functions with arity 0 are called constants. The formula $\mathsf{ite}(\tau_1 = \tau_2, \psi_1, \psi_2)$ is a shorthand for $(\tau_1 = \tau_2 \Rightarrow \psi_1) \wedge (\tau_1 \neq \tau_2 \Rightarrow \psi_2)$. Finally, Boolean-valued function applications occurring in formula positions are automatically lifted to predicate applications.

4.3. Signature

Each term in the logic has a corresponding sort. The sorts are *ref*, the sort of object references, *bool*, the sort of booleans, *set*, the sort of sets of memory locations, *fname*, the sort of field names, *cname*, the sort of class names, and *heap*, the sort of heaps. We define *val* as the union of the sorts *ref* and *bool*. We do not mention sorts when 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 are the following:

function	sort
null	*ref*
emptyset	*set*
singleton	$ref \times fname \rightarrow set$
union	$set \times set \rightarrow set$
contains	$ref \times fname \times set \rightarrow bool$
setminus	$set \times set \rightarrow set$
select	$heap \times ref \times fname \rightarrow val$
store	$heap \times ref \times fname \times val \rightarrow heap$
allocated	$heap \times ref \rightarrow bool$
allocate	$ref \times heap \times cname \rightarrow heap$
wf	$heap \rightarrow bool$
succ	$heap \times heap \rightarrow bool$

null represents the null reference. The functions *emptyset*, *singleton*, *union*, *contains* and *setminus* represent operations on sets of locations. To avoid cluttering the verification conditions, we employ the standard mathematical notation instead of explicitly writing down the operations as function invocations. For example, we write $(o, f) \in s$ instead of $contains(o, f, s)$. The heap is modeled in the verification logic as a mapping from locations to values. $select(h, o, f)$ returns the value of (o, f) in heap h and $store(h, o, f, v)$ returns a new heap that is equal to h at each location except for (o, f). More specifically, the value of (o, f) equals v in the heap $store(h, o, f, v)$. We abbreviate $select(h, o, f)$ as $h(o, f)$ and $store(h, o, f, v)$ as $h[(o, f) \mapsto v]$. $allocated(h, o)$ returns whether o is allocated in heap h. We write $\mathsf{allocated}_h(x_1, \ldots, x_n)$ as a shorthand for $(x_1 = null \vee allocated(h, x_1)) \wedge \ldots \wedge (x_n = null \vee allocated(h, x_n))$. $allocate(o, h, C)$ returns a new heap where o is allocated. $wf(h)$ denotes that the heap h is potentially reachable by the program. An important property of reachable heaps, is that fields of allocated objects only point to other allocated objects. Finally, $succ(h_1, h_2)$ denotes that heap h_2 is a successor of heap h_1. h_2 is a successor of h_1 if each object allocated in h_1 is also allocated in h_2. Moreover, a successor of a well-formed heap is well-formed as well.

The program-specific functions are the following. For each class C in the program text, the logic includes a constant C with sort *cname*. Similarly, for each field f, the logic includes a constant f with sort *fname*. A standard technique in verification is to encode pure methods as functions in the verification logic. For each pure method p in a class C with parameters $C_1\ x_1, \ldots, C_n\ x_n$ and return type t, the logic includes a function symbol $C.p$ with sort $heap \times ref \times ref_1 \times \ldots \times ref_n \rightarrow \mathsf{sort}(t)$. For example, when verifying the program of Figure 4, the logic includes a function $C.valid$ with sort $heap \times ref \rightarrow bool$. sort is a function that maps source types to sorts. More specifically, $\mathsf{sort}(C)$ equals *ref* (for each $C \in \mathcal{C}$), $\mathsf{sort}(\mathbf{set})$ equals *set* and $\mathsf{sort}(\mathbf{bool})$ equals *bool*.

4.4. Theory

We check that the generated verification conditions are valid with respect to a certain theory. $\Sigma_{prelude}$ is a theory which axiomatizes the built-in functions. For example, $\Sigma_{prelude}$ contains the following axiom:

$$\forall h, o, f, v \bullet select(store(h, o, f, v), o, f) = v$$

The above axiom encodes that the value of $o.f$ equals v after a field update $o.f := v;$. Each pure method p has a corresponding theory, Σ_p, consisting of an implementation axiom, a frame axiom and potentially an allocated axiom. More specifically, the axioms in the theory Σ_p corresponding to the pure method p in class C

pure t $p(C_1\ x_1, \ldots, C_n\ x_n)$
  requires e_{pre};
  reads e_{read};
{ **return** e_{body}; }

are the following:

- **Implementation axiom**. The implementation axiom relates the function symbol to the implementation

of the pure method.

$$\begin{gathered}\forall h, this, x_1, \ldots, x_n \bullet \\ wf(h) \wedge this \neq null \wedge \mathsf{allocated}_h(this, x_1, \ldots, x_n) \wedge \mathsf{Tr}(e_{pre}) \\ \Downarrow \\ C.p(h, this, x_1, \ldots, x_n) = \mathsf{Tr}(e_{body})\end{gathered}$$

$\mathsf{Tr}(e)$ denotes the translation of a source expression e to its encoding in first-order logic (see Section 4.5).

- **Frame axiom**. The frame axiom encodes the fact that p's return value only depends on locations in its reads clause. That is, the return value of p is equal in two heaps h_1 and h_2 if p's precondition holds in both heaps and each location (o, f) in the reads clause has the same value in h_1 and h_2.

$$\begin{gathered}\forall h_1, h_2, this, x_1, \ldots, x_n \bullet \\ wf(h_1) \wedge succ(h_1, h_2) \wedge this \neq null \wedge \mathsf{allocated}_{h_1}(this, x_1, \ldots, x_n) \wedge \\ \mathsf{Tr}(e_{pre})[h_1/h] \wedge \mathsf{Tr}(e_{pre})[h_2/h] \wedge \\ (\forall o, f \bullet (o, f) \in \mathsf{Tr}(e_{read})[h_1/h] \Rightarrow h_1(o, f) = h_2(o, f)) \\ \Downarrow \\ C.p(h_1, this, x_1, \ldots, x_n) = C.p(h_2, this, x_1, \ldots, x_n)\end{gathered}$$

- **Allocated axiom**. If the return type of p equals **set** (i.e. if p is a dynamic frame), then Σ_p includes an allocated axiom. The allocated axiom states that the set returned by a dynamic frame contains only allocated locations.

$$\begin{gathered}\forall h, this, x_1, \ldots, x_n, o, f \bullet \\ wf(h) \wedge this \neq null \wedge \mathsf{allocated}_h(this, x_1, \ldots, x_n) \wedge \mathsf{Tr}(e_{pre}) \wedge \\ (o, f) \in C.p(h, this, x_1, \ldots, x_n) \\ \Downarrow \\ allocated(h, o)\end{gathered}$$

  The allocated axiom enables the solver to prove that creation of a new object does not affect existing footprints and pure methods.

Σ_π is the union of $\Sigma_{prelude}$ and the theories of the pure methods and dynamic frames defined in π. Σ^I_{p*} is the union of the implementation axioms of all pure methods appearing before p in the program text. Note that Σ^I_{p*} does not include p's implementation axiom. Σ^F_{p*} is the union of the frame axioms of all pure methods appearing before p in the program text. Σ^I_π is the union of the implementation axioms of π's pure methods.

4.5. Verification Conditions

We check the correctness of a program by generating verification conditions. The verification conditions for each statement are shown in Figure 10. The free variables of $\mathsf{vc}(\overline{s}, \psi)$ are h, h_{old} and the free variables of ψ and $\overline{s}$. The variable h denotes the heap and h_{old} denotes the heap in the method pre-state. Tr and Df denote the translation, respectively the definedness of expressions, shown in Figures 11 and 12. The functions mpre, mmod, mbody and mpost respectively return the precondition, modifies clause, body and postcondition of a method.

The verification conditions for statements of Figure 10 are mostly standard. In the verification condition for field update, the verification condition of the remaining statements must hold under the assumption that the updated heap is a successor of h, i.e. that $succ(h, h[(\mathsf{Tr}(e_1), f) \mapsto \mathsf{Tr}(e_2)])$. The modifies clause of a method call is encoded via the conjunct $\mathsf{frame}(M)$, which is a shorthand for

$$(\forall o, f \bullet \neg allocated(h_{old}, o) \vee (o, f) \in \mathsf{Tr}(M)[h_{old}/h] \vee h_{old}(o, f) = h(o, f))$$

That is, for each location (o, f) it holds that either the location was not allocated in the call's pre-state, the location is part of the modifies clause, or its value is the same in the old and new heap. The first disjunct in the quantifier allows the callee to assign to fields of newly allocated objects without having to mention those locations in its modifies clause.

Invocations of pure methods are encoded as invocations of the corresponding functions in the verification logic. For example, the postcondition of *Iterator*'s constructor, $\mathbf{this}.getList() = l$ is encoded in the verification

$$
\begin{array}{lcl}
\mathsf{vc}(C\ x;\ \overline{s}, \psi) & ::= & \forall x \bullet \mathsf{vc}(\overline{s}, \psi) \\
\mathsf{vc}(e_1.f := e_2;\ \overline{s}, \psi) & ::= & \mathsf{Df}(e_1) \wedge \mathsf{Df}(e_2) \wedge \mathsf{Tr}(e_1) \neq \mathit{null} \wedge \\
 & & (\mathit{succ}(h, h[(\mathsf{Tr}(e_1), f) \mapsto \mathsf{Tr}(e_2)]) \Rightarrow \\
 & & \quad \mathsf{vc}(\overline{s}, \psi)[h[(\mathsf{Tr}(e_1), f) \mapsto \mathsf{Tr}(e_2)]/h]) \\
\mathsf{vc}(x := e;\ \overline{s}, \psi) & ::= & \mathsf{Df}(e) \wedge \mathsf{vc}(\overline{s}, \psi)[\mathsf{Tr}(e)/x] \\
\mathsf{vc}(e_0.m(e_1, \ldots, e_n);\ \overline{s}, \psi) & ::= & \mathsf{Df}(e_0) \wedge \ldots \wedge \mathsf{Df}(e_n) \wedge \mathsf{Tr}(e_0) \neq \mathit{null} \wedge \mathsf{Tr}(P) \wedge \\
 & & (\forall h' \bullet \\
 & & \quad \mathit{succ}(h, h') \wedge \\
 & & \quad \mathsf{Tr}(Q)[h'/h, h/h_{old}] \wedge \\
 & & \quad \mathsf{frame}(M)[h'/h, h/h_{old}] \\
 & & \quad \Rightarrow \\
 & & \quad \mathsf{vc}(\overline{s}, \psi)[h'/h] \\
 & &) \\
 & & \text{where } C \text{ is the type of } e_0,\ x_1, \ldots, x_n \\
 & & \quad \text{are the parameters of } C.m, \\
 & & P \text{ is } \mathsf{mpre}(C, m)[e_0/\mathbf{this}, e_1/x_1, \ldots, e_n/x_n] \text{ and} \\
 & & M \text{ is } \mathsf{mmod}(C, m)[e_0/\mathbf{this}, e_1/x_1, \ldots, e_n/x_n] \text{ and} \\
 & & Q \text{ is } \mathsf{mpost}(C, m)[e_0/\mathbf{this}, e_1/x_1, \ldots, e_n/x_n] \\
\mathsf{vc}(x := \mathbf{new}\ C;\ \overline{s}, \psi) & ::= & (\forall y \bullet \\
 & & \quad \neg \mathit{allocated}(h, y) \wedge y \neq \mathit{null} \wedge \\
 & & \quad \mathit{succ}(h, \mathit{allocate}(y, h, C)) \\
 & & \quad \Rightarrow \\
 & & \quad \mathsf{vc}(\overline{s}, \psi)[\mathit{allocate}(y, h, C)/h, y/x]) \\
\mathsf{vc}(\mathbf{assert}\ e;\ \overline{s}, \psi) & ::= & \mathsf{Df}(e) \wedge \mathsf{Tr}(e) \wedge \mathsf{vc}(\overline{s}, \psi) \\
\mathsf{vc}(\mathsf{nil}, \psi) & ::= & \psi
\end{array}
$$

Fig. 10. Verification conditions (vc) of statements with respect to postcondition ψ.

$$
\begin{array}{lcl}
\mathsf{Tr}(\mathit{null}) & ::= & \mathit{null} \\
\mathsf{Tr}(x) & ::= & x \\
\mathsf{Tr}(e.f) & ::= & h(e, f) \\
\mathsf{Tr}(e_0.p(e_1, \ldots, e_n)) & ::= & C.p(h, \mathsf{Tr}(e_0), \ldots, \mathsf{Tr}(e_n)) \\
\mathsf{Tr}(\mathbf{old}(e)) & ::= & \mathsf{Tr}(e)[h_{old}/h] \\
 & & \\
\mathsf{Tr}(\mathbf{true}) & ::= & \mathit{true} \\
\mathsf{Tr}(e_1 = e_2) & ::= & \mathsf{Tr}(e_1) = \mathsf{Tr}(e_2) \\
\mathsf{Tr}(e_1 \wedge e_2) & ::= & \mathsf{Tr}(e_1) \wedge \mathsf{Tr}(e_2) \\
\mathsf{Tr}(\mathsf{fresh}(e)) & ::= & (\forall o, f \bullet (o, f) \in \mathsf{Tr}(e) \Rightarrow \neg \mathit{allocated}(h_{old}, o)) \\
\mathsf{Tr}(e_0\ ?\ e_1 : e_2) & ::= & \mathsf{ite}(\mathsf{Tr}(e_0), \mathsf{Tr}(e_1), \mathsf{Tr}(e_2)) \\
 & & \\
\mathsf{Tr}(\emptyset) & ::= & \emptyset \\
\mathsf{Tr}(\{e.f\}) & ::= & \{(\mathsf{Tr}(e), f)\} \\
\mathsf{Tr}(e_1 \cup e_2) & ::= & \mathsf{Tr}(e_1) \cup \mathsf{Tr}(e_2) \\
\mathsf{Tr}(e_1 \setminus e_2) & ::= & \mathsf{Tr}(e_1) \setminus \mathsf{Tr}(e_2)
\end{array}
$$

Fig. 11. Translation of expressions to first-order logic.

logic as the formula $\mathit{Iterator.getList}(h, \mathit{this}) = l$. Note that definedness of conjunction assumes $\wedge$ shortcuts if the left-hand side is **false**. That is, the right-hand side has to be well-defined only if the translation of the left-hand side is *true*.

4.6. Validity

We are now ready to define what it means for a program to be valid. A program is *valid* if all methods and the main routine are valid (Definition 5). A mutator is valid if the precondition is well-defined,

$$
\begin{array}{lcl}
\mathsf{Df}(\mathit{null}) & ::= & \mathit{true} \\
\mathsf{Df}(x) & ::= & \mathit{true} \\
\mathsf{Df}(e.f) & ::= & \mathsf{Df}(e) \wedge \mathsf{Tr}(e) \neq \mathit{null} \\
\mathsf{Df}(e_0.p(e_1,\ldots,e_n)) & ::= & \mathsf{Df}(e_0) \wedge \ldots \wedge \mathsf{Df}(e_n) \wedge \mathsf{Tr}(e_0) \neq \mathit{null} \wedge \mathsf{Tr}(P) \\
 & & \text{where } P \text{ is } \mathsf{mpre}(C,p)[e_0/\mathbf{this}, e_1/x_1, \ldots, e_n/x_n] \\
\mathsf{Df}(\mathbf{old}(e)) & ::= & \mathsf{Df}(e)[h_{old}/h] \\
 & & \\
\mathsf{Df}(\mathbf{true}) & ::= & \mathit{true} \\
\mathsf{Df}(e_1 = e_2) & ::= & \mathsf{Df}(e_1) \wedge \mathsf{Df}(e_2) \\
\mathsf{Df}(e_1 \wedge e_2) & ::= & \mathsf{Df}(e_1) \wedge (\mathsf{Tr}(e_1) \Rightarrow \mathsf{Df}(e_2)) \\
\mathsf{Df}(\mathsf{fresh}(e)) & ::= & \mathsf{Df}(e) \\
\mathsf{Tr}(e_0 \ ?\ e_1 : e_2) & ::= & \mathsf{Df}(e_0) \wedge \mathsf{ite}(\mathsf{Tr}(e_0), \mathsf{Df}(e_1), \mathsf{Df}(e_2)) \\
 & & \\
\mathsf{Df}(\emptyset) & ::= & \mathit{true} \\
\mathsf{Df}(\{e.f\}) & ::= & \mathsf{Df}(e) \wedge \mathsf{Tr}(e) \neq \mathit{null} \\
\mathsf{Df}(e_1 \cup e_2) & ::= & \mathsf{Df}(e_1) \wedge \mathsf{Df}(e_2) \\
\mathsf{Df}(e_1 \setminus e_2) & ::= & \mathsf{Df}(e_1) \wedge \mathsf{Df}(e_2)
\end{array}
$$

Fig. 12. Definedness of expressions in first-order logic.

the modifies clause and postcondition are well-defined provided the precondition holds and the method body satisfies the method contract (Definition 2). The main routine is valid if it satisfies the contract **requires true**; **modifies** $\emptyset$; **ensures true**; (Definition 3). A pure method is valid if the precondition is well-defined, the reads clause and body are well-defined provided the precondition holds and the return value depends only on locations in the reads clause (Definition 4). Definedness of method contracts is similar to the notion of protection in [LW98].

Note that pure methods are not verified with respect to the theory Σ_π. Instead, the definedness of the precondition and body is checked in the theory $\Sigma_{prelude} \cup \Sigma^I_{p*}$. Satisfaction of the reads clause is checked with respect to the theory $\Sigma_{prelude} \cup \Sigma^I_\pi \cup \Sigma^F_{p*}$. The intuition underlying this decision is as follows. A program is verified in 4 phases. Each phase consists of a number of steps. During a step, one can only rely on assumptions that have been verified in previous phases and steps. In the first phase, we check whether the preconditions and method bodies of the pure methods of the program are well-defined from top to bottom. In each step of the first phase, one can assume that the prelude axioms hold and that the preconditions and method bodies of pure methods defined earlier in the program text are well-defined. In Section 5, we prove that the latter assumption implies that the implementation axiom holds. Therefore, the first phase uses the theory $\Sigma_{prelude} \cup \Sigma^I_{p*}$. In the second phase, we check whether the reads clauses of pure methods are well-defined. Since we already checked the well-definedness of preconditions and method bodies, we perform the latter check in the theory $\Sigma_{prelude} \cup \Sigma^I_\pi$. Phase 3 of the verification process consists of checking whether the pure methods satisfy their reads annotation. Since we proceed again from top to bottom, we can check compliance with the reads clause assuming that the prelude axioms hold, that the implementation axioms of all pure methods hold and that all pure methods defined earlier in the program text satisfy their reads annotation. That is, we perform the latter check in the theory $\Sigma_{prelude} \cup \Sigma^I_\pi \cup \Sigma^F_{p*}$. Finally, we verify the mutators. In this final phase, we can safely assume the prelude axioms and the axioms corresponding to the pure methods. That is, mutators are verified with respect to the theory Σ_π.

Definition 2. A mutator m defined in a class C

$$\textbf{void } m(C_1\ x_1, \ldots, C_n\ x_n) \textbf{ requires } e_{pre};\ \textbf{modifies } e_{mod};\ \textbf{ensures } e_{post};\ \{\ \overline{s}\ \}$$

is *valid* if all of the following hold:

- The precondition is well-defined and the modifies clause is well-defined, provided the precondition holds.

$$
\begin{array}{c}
\Sigma_\pi \vdash \forall h, \mathit{this}, x_1, \ldots, x_n \bullet \\
\mathit{wf}(h) \wedge \mathit{this} \neq \mathit{null} \wedge \mathsf{allocated}_h(\mathit{this}, x_1, \ldots, x_n) \\
\Downarrow \\
\mathsf{Df}(e_{pre}) \wedge (\mathsf{Tr}(e_{pre}) \Rightarrow \mathsf{Df}(e_{mod}))
\end{array}
$$

- The postcondition is well-defined, provided the precondition held in the method pre-state.

$$\begin{array}{c}\Sigma_\pi \vdash \forall h, h_{old}, this, x_1, \ldots, x_n \bullet \\ wf(h_{old}) \wedge succ(h_{old}, h) \wedge \\ this \neq null \wedge \mathsf{allocated}_{h_{old}}(this, x_1, \ldots, x_n) \wedge \mathsf{Tr}(\mathbf{old}(e_{pre})) \\ \Downarrow \\ \mathsf{Df}(e_{post})\end{array}$$

- The method body satisfies the method contract.

$$\begin{array}{c}\Sigma_\pi \vdash \forall h, this, x_1, \ldots, x_n \bullet \\ wf(h) \wedge this \neq null \wedge \mathsf{allocated}_h(this, x_1, \ldots, x_n) \wedge \mathsf{Tr}(e_{pre}) \\ \Downarrow \\ \mathsf{vc}(\overline{s}, \mathsf{Tr}(e_{post}) \wedge \mathsf{frame}(e_{mod}))[h/h_{old}]\end{array}$$

Definition 3. The main routine $\overline{s}$ is *valid* if the following holds:

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

Definition 4. A pure method p defined in a class C

$$\mathbf{pure}\ t\ p(C_1\ x_1, \ldots, C_n\ x_n)\ \mathbf{requires}\ e_{pre};\ \mathbf{reads}\ e_{read};\ \{\ \mathbf{return}\ e_{body};\ \}$$

is *valid* if all of the following hold:

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

$$\begin{array}{c}\Sigma_{prelude} \cup \Sigma^I_{p*} \vdash \forall h, this, x_1, \ldots, x_n \bullet \\ wf(h) \wedge this \neq null \wedge \mathsf{allocated}_h(this, x_1, \ldots, x_n) \\ \Downarrow \\ \mathsf{Df}(e_{pre}) \wedge (\mathsf{Tr}(e_{pre}) \Rightarrow \mathsf{Df}(e_{body}))\end{array}$$

- The reads clause is well-defined, provided the precondition holds.

$$\begin{array}{c}\Sigma_{prelude} \cup \Sigma^I_{\pi} \vdash \forall h, this, x_1, \ldots, x_n \bullet \\ wf(h) \wedge this \neq null \wedge \mathsf{allocated}_h(this, x_1, \ldots, x_n) \wedge \mathsf{Tr}(e_{pre}) \\ \Downarrow \\ \mathsf{Df}(e_{read})\end{array}$$

- The return value only depends on locations in the reads clause.

$$\begin{array}{c}\Sigma_{prelude} \cup \Sigma^I_{\pi} \cup \Sigma^F_{p*} \vdash \forall h_1, h_2, this, x_1, \ldots, x_n \bullet \\ wf(h_1) \wedge succ(h_1, h_2) \wedge this \neq null \wedge \\ \mathsf{allocated}_{h_1}(this, x_1, \ldots, x_n) \wedge \mathsf{Tr}(e_{pre})[h_1/h] \wedge \mathsf{Tr}(e_{pre})[h_2/h] \wedge \\ (\forall o, f \bullet (o, f) \in \mathsf{Tr}(e_{read})[h_1/h] \Rightarrow h_1(o, f) = h_2(o, f)) \\ \Downarrow \\ \mathsf{Tr}(e_{body})[h_1/h] = \mathsf{Tr}(e_{body})[h_2/h]\end{array}$$

Definition 5. A program is *valid* if the all methods and the main routine are valid.

5. Soundness

In the previous section, we defined the notion of valid program. In this section, we prove that valid programs never dereference null and assert statements never fail. We proceed as follows. We start by defining an execution semantics for the programs written in the language defined in Figure 9. This semantics performs run-time checking. More specifically, a failed assert, null dereference, or precondition/postcondition violation will cause the program to get stuck. Finally, we prove that valid programs do not get stuck. We do so by proving that the initial configuration is a *valid configuration*, that the small-step relation $\rightarrow$ preserves validity of configurations and that valid configurations are never stuck.

We define $\mathcal{O}$ as the set of object references, $\mathbb{B}$ as the set of booleans and $\mathcal{V}$ as the set of values (the union of $\mathcal{O}$ and $\mathbb{B}$). $\mathcal{O}$ includes the *null* reference.

5.1. Execution Semantics

We start by defining configurations.

Definition 6. A configuration σ consists of:

- a heap H, a partial function from object references to object states. An object state is a partial function from field names to object references.
- a stack S, a list of activation records. Each activation record consists of:
  - an environment Γ, a partial function from variable names to object references.
  - an old heap G, the heap at the time the activation was put onto the call stack.
  - a statement sequence $\overline{s}$.

An object o is *allocated* in a heap H if $o \in \mathsf{dom}(H)$. We sometimes implicitly uncurry the heap. That is, we write $h(o, f)$ instead of $h(o)(f)$. Execution of a program starts in the *initial configuration*, $(\emptyset, (\emptyset, \emptyset, \overline{s}))$, where $\overline{s}$ is the program's main routine. That is, in the initial configuration the heap is empty and the stack contains a single activation record. The environment and old heap of this activation record are both empty.

5.1.1. Statements

Execution of statements is defined by the small-step relation $\rightarrow$ defined in Figure 13. $f[a \mapsto b]$ denotes an update of the function f at a with b. That is, $f[a \mapsto b](x)$ equals $f(x)$ unless x equals a. $f[a \mapsto b](a)$ equals b. Each non-null object reference o has a corresponding type, denoted as $\mathsf{type}(o)$. The list of fields declared in class C is denoted $\mathsf{fields}(C)$.

A variable declaration $C\ x$; adds a new entry (x, null) to the environment. A variable declaration can never get stuck. A field update $e_1.f := e_2$; sets the value of the heap at location (v_1, f) to v_2, provided e_1 evaluates to v_1 and e_2 to v_2. A field update gets stuck if either e_1 or e_2 gets stuck or if v_1 is *null*. A variable update $x := e$; updates the value of x in the environment. A variable update gets stuck if evaluation of e gets stuck. A mutator invocation $e_0.m(e_1, \ldots, e_n)$ adds a new activation record to the call stack that executes m's body. A mutator invocation is stuck if one of the arguments is stuck, if e_0 evaluates to *null* or if the precondition does not hold. When the list of statements in the top activation record is empty, the top activation record is popped from the stack. A return is stuck if the postcondition does not hold or if more locations were modified than allowed by the modifies clause. An object creation allocates a fresh object reference o. More specifically, a new object state is added to the heap at o. This object state maps each field of o to *null*. An object creation cannot get stuck. Finally, an assert statement **assert** e; checks whether e evaluates to **true**. If it does, then the assert statement is equivalent to skip; otherwise, the statement is stuck.

Definition 7. A configuration σ is *final* (denoted $\mathsf{final}(\sigma)$) if the stack contains a single activation record with an empty statement sequence.

Definition 8. A configuration σ is *stuck* (denoted $\mathsf{stuck}(\sigma)$) if σ is not final and there exists no σ' such that $\sigma \rightarrow \sigma'$.

5.1.2. Expressions

Evaluation of an expressions is defined via a big-step semantics. More specifically, the judgment $H, G, \Gamma \vdash e \Downarrow v$ states that the expression e evaluates to v in a context where H is the heap, G is the old heap and Γ is the environment (see Figure 14).

Values have been omitted from Figure 14 as they evaluate to themselves. A variable x evaluates to its value in the environment Γ. We assume the type system guarantees that a variable cannot be stuck. Evaluating a field read $e.f$ corresponds to looking up the value at location (v, f) in the heap, provided e evaluates to v. Evaluation of a field access is stuck if evaluation of e is stuck or if v equals *null*. The value of a pure method invocation $e_0.p(e_1, \ldots, e_n)$ is the value that results from evaluating p's method body. An invocation of a pure method is stuck if evaluation of one of the arguments is stuck, if the receiver is null, if the precondition does not hold or if evaluation of p's body is stuck. To evaluate an old expression $\mathbf{old}(e)$, one must evaluate e in the old heap G. An equality $e_1 = e_2$ evaluates to **true** if the value of e_1 equals the

$$(H, (\Gamma, G, C\ x;\ \overline{s}) \cdot S) \rightarrow (H, (\Gamma[x \mapsto null], G, \overline{s}) \cdot S)$$

$$\frac{H, H, \Gamma \vdash e_1 \Downarrow v_1 \qquad H, H, \Gamma \vdash e_2 \Downarrow v_2 \qquad v_1 \neq null}{(H, (\Gamma, G, e_1.f := e_2;\ \overline{s}) \cdot S) \rightarrow (H[(v_1, f) \mapsto v_2], (\Gamma, G, \overline{s}) \cdot S)}$$

$$\frac{H, H, \Gamma \vdash e \Downarrow v}{(H, (\Gamma, G, x := e;\ \overline{s}) \cdot S) \rightarrow (H, (\Gamma[x \mapsto v], G, \overline{s}) \cdot S)}$$

$$\frac{\begin{array}{c} H, H, \Gamma \vdash e_0 \Downarrow v_0\ \ldots\ H, H, \Gamma \vdash e_n \Downarrow v_n \\ v_0 \neq null \qquad \mathsf{type}(v_0) = C \qquad \mathsf{mbody}(C.m) = \overline{r} \qquad \mathsf{mparams}(C.m) = C_1\ x_1, \ldots, C_n\ x_n \\ \Gamma' = [\mathbf{this} \mapsto v_0, x_1 \mapsto v_1, \ldots x_n \mapsto v_n] \qquad H, H, \Gamma' \vdash \mathsf{mpre}(C.m) \Downarrow \mathbf{true} \end{array}}{(H, (\Gamma, G, e_0.m(e_1, \ldots, e_n);\ \overline{s}) \cdot S) \rightarrow \qquad (H, (\Gamma', H, \overline{r}) \cdot (\Gamma, G, e_0.m(e_1, \ldots, e_n);\ \overline{s}) \cdot S)}$$

$$\frac{\begin{array}{c} \mathsf{type}(\Gamma'(\mathbf{this})) = C \qquad H, G', \Gamma' \vdash \mathsf{mpost}(C.m) \Downarrow \mathbf{true} \\ G', G', \Gamma' \vdash \mathsf{mmod}(C.m) \Downarrow s \qquad \forall (o, f) \bullet o \notin \mathsf{dom}(G') \vee (o, f) \in s \vee H(o, f) = G'(o, f) \end{array}}{(H, (\Gamma', G', \mathsf{nil}) \cdot (\Gamma, G, e_0.m(e_1, \ldots, e_n);\ \overline{s}) \cdot S) \rightarrow (H, (\Gamma, G, \overline{s}) \cdot S)}$$

$$\frac{o \neq null \qquad \mathsf{type}(o) = C \qquad o \notin \mathsf{dom}(H) \qquad \mathsf{fields}(C) = C_1\ f_1, \ldots, C_n\ f_n}{(H, (\Gamma, G, x := \mathbf{new}\ C;\ \overline{s}) \cdot S) \rightarrow \qquad (H[(o, f_1) := null, \ldots, (o, f_n) := null], (\Gamma[x \mapsto o], G, \overline{s}) \cdot S)}$$

$$\frac{H, H, \Gamma \vdash e \Downarrow \mathbf{true}}{(H, (\Gamma, G, \mathbf{assert}\ e;\ \overline{s}) \cdot S) \rightarrow (H, (\Gamma, G, \overline{s}) \cdot S)}$$

Fig. 13. Execution of statements.

$$H, G, \Gamma \vdash x \Downarrow \Gamma(x) \qquad \frac{H, G, \Gamma \vdash e \Downarrow v \qquad v \neq null}{H, G, \Gamma \vdash e.f \Downarrow H(v, f)}$$

$$\frac{\begin{array}{c} H, G, \Gamma \vdash e_0 \Downarrow v_0\ \ldots\ H, G, \Gamma \vdash e_n \Downarrow v_n \\ v_0 \neq null \qquad \mathsf{type}(v_0) = C \qquad \mathsf{mparams}(C.p) = C_1\ x_1, \ldots, C_n\ x_n \\ \Gamma' = [\mathbf{this} \mapsto v_0, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \qquad H, H, \Gamma' \vdash \mathsf{mpre}(C.p) \Downarrow \mathbf{true} \qquad H, G, \Gamma' \vdash \mathsf{mbody}(C.p) \Downarrow v \end{array}}{H, G, \Gamma \vdash e_0.p(e_1, \ldots, e_n) \Downarrow v}$$

$$\frac{G, G, \Gamma \vdash e \Downarrow v}{H, G, \Gamma \vdash \mathbf{old}(e) \Downarrow v} \qquad \frac{H, G, \Gamma \vdash e_1 \Downarrow v_1 \qquad H, G, \Gamma \vdash e_2 \Downarrow v_2}{H, G, \Gamma \vdash e_1 = e_2 \Downarrow v_1 = v_2} \qquad \frac{H, G, \Gamma \vdash e_1 \Downarrow \mathbf{false}}{H, G, \Gamma \vdash e_1 \wedge e_2 \Downarrow \mathbf{false}}$$

$$\frac{H, G, \Gamma \vdash e_1 \Downarrow \mathbf{true} \qquad H, G, \Gamma \vdash e_2 \Downarrow b_2}{H, G, \Gamma \vdash e_1 \wedge e_2 \Downarrow b_2} \qquad \frac{H, G, \Gamma \vdash e \Downarrow s \qquad b = \forall (o, f) \in s \bullet o \notin \mathsf{dom}(G)}{H, G, \Gamma \vdash \mathbf{fresh}(e) \Downarrow b}$$

$$\frac{H, G, \Gamma \vdash e_0 \Downarrow \mathbf{true} \qquad H, G, \Gamma \vdash e_1 \Downarrow v}{H, G, \Gamma \vdash e_0\ ?\ e_1 : e_2 \Downarrow v} \qquad \frac{H, G, \Gamma \vdash e_0 \Downarrow \mathbf{false} \qquad H, G, \Gamma \vdash e_2 \Downarrow v}{H, G, \Gamma \vdash e_0\ ?\ e_1 : e_2 \Downarrow v}$$

$$\frac{H, G, \Gamma \vdash e \Downarrow v}{H, G, \Gamma \vdash \{\ e.f\ \} \Downarrow \{\ (v, f)\ \}} \qquad \frac{H, G, \Gamma \vdash e_1 \Downarrow s_1 \qquad H, G, \Gamma \vdash e_2 \Downarrow s_2}{H, G, \Gamma \vdash e_1 \cup e_2 \Downarrow s_1 \cup s_2}$$

$$\frac{H, G, \Gamma \vdash e_1 \Downarrow s_1 \qquad H, G, \Gamma \vdash e_2 \Downarrow s_2}{H, G, \Gamma \vdash e_1 \setminus e_2 \Downarrow s_1 \setminus s_2}$$

Fig. 14. Evaluation of expressions.

value of e_2. An equality is stuck if either the left or right-hand side is stuck. A conjunction $e_1 \wedge e_2$ evaluates to **false** if e_1 evaluates to **false**; otherwise, the conjunction evaluates to the value of e_2. Note that this is the shortcut semantics for conjunction: the conjunction is stuck if the left-hand side is stuck or if the left-hand side evaluates to **true** and the right-hand side is stuck. A fresh expression **fresh**(e) evaluates to **true** if all locations in e's value were not allocated in the old heap G. A fresh expression is stuck only if evaluation of e is stuck. A conditional expression $e_0 \ ? \ e_1 : e_2$ evaluates to the value of e_1 if e_0 evaluates to **true**; otherwise, the conditional expression evaluates to the value of e_2. A conditional expression is stuck if e_0 is stuck, if e_0 evaluates to **true** and e_1 is stuck, or if e_0 evaluates to **false** and e_2 is stuck. A singleton $\{\ e.f\ \}$ evaluates to the singleton set $\{\ (v, f)\ \}$, provided v is the value of e. A singleton expression is stuck if e is stuck. A union $e_1 \cup e_2$ evaluates to the union of the values of e_1 and e_2. The union is stuck if either e_1 or e_2 is stuck. The semantics of subtraction is similar to union.

Definition 9. Evaluation of an expression e is *stuck* in a heap H, old heap G and environment Γ if there does not exists a value v such that $H, G, \Gamma \vdash e \Downarrow v$.

5.1.3. Well-formed Configurations

Executions of well-formed programs satisfy certain well-formedness properties.

Definition 10. A heap H is *well-formed* (denoted $\mathsf{wf}(H)$) if fields of allocated objects never point to non-allocated objects.

$$\mathsf{wf}(H) \Leftrightarrow \forall o, f \bullet o \in \mathsf{dom}(H) \wedge H(o, f) \neq \mathit{null} \Rightarrow H(o, f) \in \mathsf{dom}(H)$$

Definition 11. An environment Γ is *well-formed* with respect to a heap H (denoted $\mathsf{wf}(\Gamma, H)$) if all non-null object references in the range of Γ are allocated in H.

$$\mathsf{wf}(\Gamma, H) \Leftrightarrow \forall x \in \mathsf{dom}(\Gamma) \bullet \Gamma(x) = \mathit{null} \vee \Gamma(x) \in \mathsf{dom}(H)$$

Definition 12. An heap H' is a *successor* of a heap H (denoted $\mathsf{succ}(H, H')$) if locations allocated in H are still allocated in H'.

$$\mathsf{succ}(H, H') \Leftrightarrow \mathsf{dom}(H) \subseteq \mathsf{dom}(H')$$

A heap H is a predecessor of H' if H' is a successor of H.

Definition 13. A configuration $\sigma = (H, (\Gamma_1, G_1, \overline{s_1}) \cdot \ldots \cdot (\Gamma_k, G_k, \overline{s_k}))$ is *well-formed* (denoted $\mathsf{wf}(\sigma)$) if all of the following hold:

1. The heap H is well-formed.
2. For each activation record $(\Gamma_i, G_i, \overline{s_i})$ with $i \in \{1, \ldots, n\}$ the following hold:
   (a) G_i is a well-formed heap. Moreover, G_i is a predecessor of G_{i-1} (or of H if i equals 1).
   (b) The environment Γ_i is well-formed wrt G_{i-1} (or wrt H if i equals 1).
   (c) The free variables of $\overline{s_i}$ are in the domain of Γ_i.
   (d) If i is not equal to 1, then the first statement in the activation record is a method invocation $e_0.m(e_1, \ldots, e_j)$.

Before we can prove that $\rightarrow$ preserves well-formedness of configurations, we need to show that evaluation of an expression that does not contain old subexpressions never yields a non-null, non-allocated object reference.

Theorem 1. Suppose H and G are well-formed heaps and Γ is a well-formed environment with respect to H. Suppose e is an expression that does not contain old subexpressions and whose free variables are in the domain of Γ. If evaluation of e yields a non-null object reference, then that object reference is allocated.

$$\forall H, G, \Gamma, e \bullet \mathsf{wf}(H) \wedge \mathsf{wf}(G) \wedge \mathsf{wf}(\Gamma, H) \wedge \mathsf{free}(e) \subseteq \mathsf{dom}(\Gamma) \wedge \\ H, G, \Gamma \vdash e \Downarrow v \wedge v \in \mathcal{O} \Rightarrow v = \mathit{null} \vee v \in \mathsf{dom}(H)$$

Proof. By case analysis on the type of expression and induction on the depth of the derivation.

- **Constant**. Suppose the expression is *null*. Trivial.

- **Variable**. Suppose the expression is a variable x. Since Γ is well-formed with respect to H, it follows that $\Gamma(x)$ is never a non-allocated, non-null reference.
- **Field read**. Suppose the expression is a field read $e.f$. Since the derivation for e is smaller than the derivation for $e.f$, it follows that the value of e, say v, is allocated in H. Since H is well-formed, $H(v, f)$ is allocated in H.
- **Method invocation**. Suppose the expression is a method invocation $e_0.p(e_1, \ldots, e_n)$. Since derivation of each e_i is smaller than $e_0.p(e_1, \ldots, e_n)$, it follows that the value of an e_i, say v_i, is never a non-null, non-allocated reference. Therefore, $[\textbf{this} \mapsto v_0, \ldots, x_n \mapsto v_n]$ is a well-formed environment wrt H. Suppose the body of p is **return** e_{body};. The derivation of e_{body} is smaller than the derivation of $e_0.p(e_1, \ldots, e_n)$, hence the result is never a non-null, non-allocated object reference.

□

We do not need to consider other expression types, since the cases considered above are the only expressions that evaluate to object references.

Theorem 2. Suppose H and G are well-formed heaps, Γ is a well-formed environment with respect to H and H is a successor of G. Suppose e is an expression that does not contain old subexpressions and whose free variables are in the domain of Γ. If evaluation of an expression e yields a set of memory locations, then that set contains only allocated object references.

Proof. By case analysis on the type of expression and induction on the depth of the derivation. □

Theorem 3. $\rightarrow$ preserves well-formedness of configurations.

$$\forall \sigma, \sigma' \bullet \mathsf{wf}(\sigma) \wedge \sigma \rightarrow \sigma' \Rightarrow \mathsf{wf}(\sigma')$$

Proof. Suppose $\sigma = (H, (\Gamma_1, G_1, \overline{s_1}) \cdot \ldots \cdot (\Gamma_k, G_k, \overline{s_k}))$ is a well-formed configuration. By case analysis on the first statement of $\overline{s_1}$.

- **Variable declaration** Suppose the first statement of $\overline{s_1}$ is a variable declaration $C\ x$;. Trivial.
- **Variable update**. Suppose the first statement of $\overline{s_1}$ is a variable update $x := e$;. It follows immediately from Theorem 1 that the environment remains well-formed.
- **Field update**. Suppose the first statement of $\overline{s_1}$ is a field update $e_1.f := e_2$;. It follows from Theorem 1 that H remains well-formed.
- **Mutator invocation**. Suppose the first statement of $\overline{s_1}$ is a mutator invocation $e_0.m(e_1, \ldots, e_n)$. The old top activation record remains well-formed. The new old heap is well-formed since the heap is well-formed. The new environment is well-formed because of Theorem 1. Because of the well-formedness of the program, the free variables of the statements in the new top activation record are in the new environment. The successor relation is reflexive so the new old heap, H, is a predecessor of H.
- **Return**. Suppose $\overline{s_1}$ is the empty sequence. The successor relation is transitive so the global heap is a successor of the old heap of the new top activation record.
- **Object Creation**. Suppose the first statement of $\overline{s_1}$ is an object creation $x :=$ **new** C;. The environment remains well-formed, since x refers to an allocated object. The heap remains well-formed, because the fields of the new object all point to *null*. The heap remains a successor of G_1.
- **Assert**. Suppose the first statement of $\overline{s_1}$ is an assert statement **assert** e;. The assert statement is equivalent to skip if e evaluates to **true**.

□

Theorem 4. Executions of well-formed programs reach only well-formed configurations.

Proof. Follows immediately from the well-formedness of the initial configuration and Theorem 3. □

5.2. Interpretation

We interpret the verification logic using the interpretation $\mathfrak{I}$. First of all, $\mathfrak{I}$ maps each sort to a set.

sort	set
val	$\mathcal{V} = \mathcal{O} \cup \mathbb{B}$
ref	$\mathcal{O}$
bool	$\mathbb{B}$
set	$\mathcal{P}(\mathcal{O} \times \mathcal{F})$
cname	$\mathcal{C}$
fname	$\mathcal{F}$
heap	$\mathcal{O} \times \mathcal{F} \to \mathcal{V}$

Secondly, $\mathfrak{I}$ maps each function symbol in the signature to a function. The interpretation for the built-in functions is as expected. We assume $\mathfrak{I}$ is a model for the prelude axioms. That is, $\mathfrak{I} \models \Sigma_{prelude}$.

function	interpretation
$null$	$null$
$emptyset$	$\emptyset$
$singleton(o, f)$	$\{\ (o, f)\ \}$
$union(s_1, s_2)$	$s_1 \cup s_2$
$contains(s, o, f)$	$(o, f) \in s$
$setminus(s_1, s_2)$	$s_1 \setminus s_2$
$select(h, o, f)$	$h(o, f)$
$store(h, o, f, v)$	$h[(o, f) \mapsto v]$
$allocated(h, o)$	$o \in \mathsf{dom}(h)$
$allocate(o, h, C)$	$h[(o, f_1) \mapsto null, \ldots, (o, f_n) \mapsto null]$ where $\mathsf{fields}(\mathfrak{I}(C)) = C_1\ f_1, \ldots, C_n\ f_n$
$wf(h)$	$\mathsf{wf}(h)$
$succ(h_1, h_2)$	$\mathsf{succ}(h_1, h_2)$

Finally, we interpret the program-specific functions as follows. The constant C is interpreted as the class name C. Similarly, the constant f is interpreted as the field name f. $\mathfrak{I}(C.p)(H, v_0, \ldots, v_n)$ equals v if there exists a v such that $v_0.p(v_1, \ldots, v_n)$ evaluates to v in heap H (i.e. $H, H, [\mathbf{this} \mapsto v_0, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \vdash \mathbf{this}.p(x_1, \ldots, x_n) \Downarrow v$); otherwise, the result equals the default value of p's return type.

We write $\mathfrak{I}, H, G, \Gamma \models \psi$ to indicate the formula ψ holds under a certain interpretation. More specifically, $\mathfrak{I}, H, G, \Gamma \models \psi$ holds if ψ is true under an interpretation that maps each function to its interpretation in $\mathfrak{I}$, that maps h to H, h_{old} to G and each other variable to its value in Γ.

5.3. Truth of Axioms

Before we can start proving soundness, we have to show that the validity of a program π implies that the corresponding theory Σ_π is true under $\mathfrak{I}$ (Theorem 6).

Theorem 5 shows that evaluation of well-defined (according to Df) expressions does not get stuck and that translation (Tr) matches evaluation. A pure method p' is larger than p, if p' appears after p in the program text.

Theorem 5. Suppose π is a valid program, H and G are well-formed heaps such that H is a successor of G, Γ is a well-formed environment with respect to H and e is an expression whose free variables are in the domain of Γ. Moreover, assume that the values of free variables occurring in old subexpressions of e are allocated in G. Finally, suppose that the largest pure method that appears in e is p and that $\mathfrak{I} \models \Sigma^I_{p*}$. If $\mathfrak{I}, H, G, \Gamma \models \mathsf{Df}(e)$, then evaluation of e does not get stuck. Moreover, the interpretation of e's translation equals the value resulting from its evaluation: $H, G, \Gamma \vdash e \Downarrow v \Rightarrow \mathfrak{I}, H, G, \Gamma \models \mathsf{Tr}(e) = v$.

Proof. By induction on the size of e and case analysis on the kind of expression. An expression e is smaller than e' if e's largest method is smaller than e''s largest method or if they have the same largest method and e is syntactically smaller than e'.

- **Null**. Suppose the expression is *null*. Evaluation of a value is never stuck. By definition of Tr and $\mathfrak{I}$, the interpretation of the translation equals the value.

- **Variable**. Suppose the expression is a variable x. Evaluation of a variable is never stuck. The expression x evaluates to $\Gamma(x)$. $\mathsf{Tr}(x) = x$ and the interpretation of a variable equals its value in the environment.
- **Field read**. Suppose the expression is a field read $e.f$. Since $e.f$ is well-defined, it follows that $\mathfrak{I}, H, G, \Gamma \models \mathsf{Df}(e) \wedge \mathsf{Tr}(e) \neq \mathit{null}$. Because e is syntactically smaller than $e.f$ and e is well-defined, the evaluation of e does not get stuck and its value, say v, equals the interpretation of its translation. That is, $\mathfrak{I}, H, G, \Gamma \models \mathsf{Tr}(e) = v$. Moreover, it follows from $\mathfrak{I}, H, G, \Gamma \models \mathsf{Tr}(e) \neq \mathit{null}$ that v is not *null*. Since evaluation of e does not get stuck and e's value is not *null*, evaluation of $e.f$ is not stuck. It immediately follows from the definition of $\mathfrak{I}$ and Tr that the value $H(v, f)$ equals the interpretation of the translation $H(\mathsf{Tr}(e), f)$.
- **Method invocation**. Suppose the expression is an invocation of a pure method $e_0.p(e_1, \ldots, e_n)$. Since $e_0.p(e_1, \ldots, e_n)$ is well-defined, it follows that $\mathfrak{I}, H, G, \Gamma \models \mathsf{Df}(e_0) \wedge \ldots \wedge \mathsf{Df}(e_n) \wedge \mathsf{Tr}(e_0) \neq \mathit{null} \wedge \mathsf{Tr}(P)$, where P is the precondition of p with e_0 substituted for **this**, e_1 for x_1, etc. Because each parameter e_i with $i \in \{0, \ldots, n\}$ is smaller than $e_0.p(e_1, \ldots, e_n)$ and each e_i is well-defined, the evaluation of e_i does not get stuck and its value, say v_i, equals the interpretation of the translation. That is, it holds for each e_i that $\mathfrak{I}, H, G, \Gamma \models \mathsf{Tr}(e_i) = v_i$. It immediately follows from $\mathfrak{I}, H, G, \Gamma \models \mathsf{Tr}(e_0) \neq \mathit{null}$ that v_0 is not *null*. π is a valid program, hence p is a valid pure method. By Definition 4, we have

$$\begin{array}{c} \Sigma_{prelude} \cup \Sigma^I_{p*} \vdash \forall h, \mathit{this}, x_1, \ldots, x_n \bullet \\ \mathit{wf}(h) \wedge \mathit{this} \neq \mathit{null} \wedge \mathsf{allocated}_h(\mathit{this}, x_1, \ldots, x_n) \Rightarrow \\ \mathsf{Df}(e_{pre}) \end{array}$$

  $\mathfrak{I}$ is a model for $\Sigma_{prelude}$ and Σ^I_{p*}. Since H is well-formed, v_0 is not *null* and each v_i is allocated it follows from Theorem 1 that $\mathfrak{I}, H, G, \Gamma' \models \mathsf{Df}(e_{pre})$, where Γ' equals $[\mathbf{this} \mapsto v_0, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$. The well-formedness of the program implies that p's precondition can only call pure methods smaller than p, and hence p's precondition is smaller than $e_0.p(e_1, \ldots, e_n)$. Therefore, the evaluation of the precondition does not get stuck and its value equals the interpretation of the translation. Since $\mathfrak{I}, H, G, \Gamma \models \mathsf{Tr}(P)$, the precondition evaluates to **true**. Therefore, evaluation of $e_0.p(e_1, \ldots, e_n)$ does not get stuck. By definition of $\mathfrak{I}$, $\mathfrak{I}, H, G, \Gamma \models \mathsf{Tr}(e_0.p(e_1, \ldots, e_n)) = v$, where $H, G, \Gamma \vdash e_0.p(e_1, \ldots, e_n) \Downarrow v$.
- **Old**. Suppose the expression is $\mathbf{old}(e)$. Since $\mathbf{old}(e)$ is well-defined, it follows that $\mathfrak{I}, H, G, \Gamma \models \mathsf{Df}(e)[h_{old}/h]$. Therefore, $\mathfrak{I}, G, G, \Gamma \models \mathsf{Df}(e)$. The free variables of e are allocated in G. Therefore, we can restrict Γ to the domain of G: $\mathfrak{I}, G, G, \Gamma_{|\mathsf{dom}(G)} \models \mathsf{Df}(e)$. Because e is syntactically smaller than $\mathbf{old}(e)$ and e is well-defined, there exists a value v such that $G, G, \Gamma_{|\mathsf{dom}(G)} \vdash e \Downarrow v$ and $\mathfrak{I}, G, G, \Gamma_{|\mathsf{dom}(G)} \models \mathsf{Tr}(e) = v$. Since the evaluation of e and the interpretation of $\mathsf{Tr}(e)$ do not depend on the values of variables that do not occur in e, it follows that $G, G, \Gamma \vdash e \Downarrow v$ and $\mathfrak{I}, G, G, \Gamma \models \mathsf{Tr}(e) = v$. Hence, $\mathfrak{I}, H, G, \Gamma \models \mathsf{Tr}(\mathbf{old}(e)) = v$.

□

Theorem 6. If π is a valid program, then $\mathfrak{I}$ is a model for Σ_π.

Proof. Assume π is a valid program. It follows from the definition of $\mathfrak{I}$ that $\mathfrak{I} \models \Sigma_{prelude}$. Therefore, it suffices to prove that $\mathfrak{I}$ is a model for the axiomatization of each pure method of π. Assume S is a subset of $\Sigma_\pi \setminus \Sigma_{prelude}$ such that the following constraints hold.

1. If S contains the implementation axiom of a pure method p, then S also contains the implementations axioms of all pure methods defined before p.
2. If S contains the frame axiom of a pure method p, then S also contains the implementations axioms of all pure methods in π and the frame axioms of all pure methods defined before p.
3. If S contains the allocated axiom of a pure method p, then S also contains the implementations and frame axioms of all pure methods in π and the allocated axioms of all pure methods defined before p.

We proceed by induction on the size of S. If S is empty, then trivially $\mathfrak{I} \models S$. Suppose S is not empty and that $\mathfrak{I} \models S$. We must show that if we extend S with an axiom α in accordance to the rules described above, then $\mathfrak{I} \models S \cup \{\alpha\}$.

- **Implementation axiom**. Suppose S does not contain all implementation axioms. Select the smallest pure method p such that S does not contain p's implementation axiom. By induction, we may assume

that $\mathfrak{I}$ is a model for S, i.e. $\mathfrak{I} \models \Sigma^I_{p*}$. We have to prove that

$$\begin{gathered}\mathfrak{I} \models \forall h, this, x_1, \ldots, x_n \bullet \\ wf(h) \wedge this \neq null \wedge \mathsf{allocated}_h(this, x_1, \ldots, x_n) \wedge \mathsf{Tr}(e_{pre}) \\ \Downarrow \\ C.p(h, this, x_1, \ldots, x_n) = \mathsf{Tr}(e_{body})\end{gathered}$$

Pick an arbitrary heap H and object references $v_0, \ldots, v_n$ such that

$$\mathfrak{I}, H, H, \Gamma \models wf(h) \wedge this \neq null \wedge \mathsf{allocated}_h(this, x_1, \ldots, x_n) \wedge \mathsf{Tr}(e_{pre})$$

where $\Gamma = [\mathbf{this} \mapsto v_0, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$. We have to prove that $\mathfrak{I}, H, H, \Gamma \models C.p(h, this, x_1, \ldots, x_n) = \mathsf{Tr}(e_{body})$. It follows from Theorem 5 that there exists a value v such that $H, H, \Gamma \vdash \mathbf{this}.p(x_1, \ldots, x_n) \Downarrow v$. Moreover, $\mathfrak{I}, H, H, \Gamma \models \mathsf{Tr}(\mathbf{this}.p(x_1, \ldots, x_n)) = v$. Since evaluating a method call corresponds to evaluating the method body, we know that $H, H, \Gamma \vdash e_{body} \Downarrow v$, where e_{body} is the body of p. Since π is a valid program, p is a valid pure method. Because $\mathfrak{I} \models \Sigma^I_{p*}$, it follows that $\mathfrak{I}, H, H, \Gamma \models \mathsf{Df}(e_{body})$. It follows from Theorem 5 that evaluation of e_{body} does not get stuck and yields a value w. Furthermore, $\mathfrak{I}, H, H, \Gamma \models \mathsf{Tr}(e_{body}) = w$. Since v equals w, $\mathfrak{I}, H, H, \Gamma \models C.p(h, this, x_1, \ldots, x_n) = \mathsf{Tr}(e_{body})$.

- **Frame axiom**. Suppose S contains the implementation axioms of all pure methods in π and the frame axioms of the pure methods smaller than p. By induction, we may assume that $\mathfrak{I}$ is a model for S, i.e. $\mathfrak{I} \models \Sigma^I_\pi \cup \Sigma^F_{p*}$. We have to prove that $\mathfrak{I}$ is a model for p's frame axiom. Since π is a valid program, p is a valid method. The validity of p implies that

$$\begin{gathered}\Sigma_{prelude} \cup \Sigma^I_\pi \cup \Sigma^F_{p*} \vdash \forall h_1, h_2, this, x_1, \ldots, x_n \bullet \\ wf(h_1) \wedge succ(h_1, h_2) \wedge this \neq null \wedge \\ \mathsf{allocated}_{h_1}(this, x_1, \ldots, x_n) \wedge \mathsf{Tr}(e_{pre})[h_1/h] \wedge \mathsf{Tr}(e_{pre})[h_2/h] \wedge \\ (\forall o, f \bullet (o, f) \in \mathsf{Tr}(e_{read})[h_1/h] \Rightarrow h_1(o, f) = h_2(o, f)) \\ \Downarrow \\ \mathsf{Tr}(e_{body})[h_1/h] = \mathsf{Tr}(e_{body})[h_2/h]\end{gathered}$$

  Since $\mathfrak{I} \models \Sigma_{prelude} \cup \Sigma^I_\pi \cup \Sigma^F_{p*}$, it follows immediately that $\mathfrak{I}$ is a model for p's frame axiom.
- **Allocated axiom**. Suppose S contains the implementation and frame axioms of all pure methods in π and the allocated axioms of the pure methods smaller than p. It follows immediately from Theorems 5 and 2 that p's allocated axiom holds.

□

Since Σ_π has a model, namely $\mathfrak{I}$, it is a consistent theory.

5.4. Soundness

We prove soundness as follows. We start by defining the notion of valid configuration. Informally, a configuration is valid if the verification condition of each activation record holds when interpreted in the activation record's environment and heap. We then prove that the initial configuration is valid, that $\rightarrow$ preserves validity of configurations and that valid configurations are never stuck.

Definition 14. A configuration $\sigma = (H, (\Gamma_1, G_1, \overline{s_1}) \cdot \ldots \cdot (\Gamma_k, G_k, \overline{s_k}))$ is *valid* (denoted $\mathsf{valid}(\sigma)$) if all of the following hold:

- σ is well-formed.
- The verification condition of the top activation record holds in the current heap H, old heap G_1 and environment Γ_1. e_{post_1} and e_{mod_1} denote the postcondition and modifies clause of the method currently being executed in the top activation record.

$$\mathfrak{I}, H, G_1, \Gamma_1 \models \mathsf{vc}(\overline{s_1}, \mathsf{Tr}(e_{post_1}) \wedge \mathsf{frame}(\mathsf{Tr}(e_{mod_1})))$$

- For each $i \in \{2, \ldots, k\}$, the verification condition of $\overline{s_i}$ holds in the heap G_{i-1}, old heap G_i and environment Γ_i with respect to postcondition e_{post_i} and modifies clause e_{mod_i}. e_{post_i} and e_{mod_i} denote the postcondition and modifies clause of the method currently being executed in the ith activation record.

$$\mathfrak{I}, G_{i-1}, G_i, \Gamma_i \models \mathsf{vc}(\overline{s_i}, \mathsf{Tr}(e_{post_i}) \wedge \mathsf{frame}(\mathsf{Tr}(e_{mod_i})))$$

Theorem 7 (Progress). If a configuration σ is valid, then it is either final or there exists a configuration σ' such $\sigma \rightarrow \sigma'$.

$$\forall \sigma \bullet \mathsf{valid}(\sigma) \Rightarrow \mathsf{final}(\sigma) \vee (\exists \sigma' \bullet \sigma \rightarrow \sigma')$$

Proof. By case analysis on the type of the first statement of the top activation record.

- **Variable declaration**. Suppose the statements in the top activation record of σ are $C\ x;\ \overline{s}$. A variable declaration cannot be stuck.
- **Variable update**. Suppose the statements in the top activation record of σ are $x := e;\ \overline{s}$. Since σ is a valid configuration, it follows that $\mathfrak{I}, H, G_1, \Gamma_1 \models \mathsf{Df}(e)$. It follows from Theorem 5 that e is not stuck and hence that $x := e;$ is not stuck.
- **Field update**. Suppose the statements in the top activation record of σ are $e_1.f := e_2;\ \overline{s}$. Since σ is a valid configuration, it follows that $\mathfrak{I}, H, G_1, \Gamma_1 \models \mathsf{Df}(e_1) \wedge \mathsf{Df}(e_2) \wedge \mathsf{Tr}(e_1) \neq \mathit{null}$. It follows from Theorem 5 that e_1 and e_2 are not stuck and that e_1's value is not *null*. Therefore, $e_1.f := e_2;$ is not stuck.
- **Mutator invocation**. Suppose the statements in the top activation record of σ are $e_0.m(e_1, \ldots, e_n);\ \overline{s}$. Since σ is a valid configuration, it follows that $\mathsf{Df}(e_0) \wedge \ldots \wedge \mathsf{Df}(e_n) \wedge \mathsf{Tr}(e_0) \neq \mathit{null} \wedge \mathsf{Tr}(P)$. It follows from Theorem 5 that evaluation of e_i (with $i \in \{0, \ldots, n\}$) does not get stuck and yields some value, say v_i. Moreover, v_0 is not *null*. Finally, the validity of π implies that m is a valid mutator. Validity of m implies that

  $$\Sigma_\pi \vdash \forall h, \mathit{this}, x_1, \ldots, x_n \bullet \\ \mathit{wf}(h) \wedge \mathit{this} \neq \mathit{null} \wedge \mathsf{allocated}_h(\mathit{this}, x_1, \ldots, x_n) \Rightarrow \mathsf{Df}(e_{pre})$$

  Because $\mathfrak{I} \models \Sigma_\pi$, it follows that

  $$\mathfrak{I}, H, H, \Gamma \models \mathit{wf}(h) \wedge \mathit{this} \neq \mathit{null} \wedge \mathsf{allocated}_h(\mathit{this}, x_1, \ldots, x_n) \Rightarrow \mathsf{Df}(e_{pre})$$

  where $\Gamma = [\mathbf{this} \mapsto v_0, x_1 \mapsto v_1, \ldots x_n \mapsto v_n]$. Since σ is a valid configuration, the heap H is well-formed. We already showed that v_0 is not *null*. Theorem 1 implies that each v_i is either *null* or allocated. It follows that $\mathfrak{I}, H, H, \Gamma \models \mathsf{Df}(P)$. Therefore, the precondition of m is not stuck. Moreover, the precondition evaluates to **true**, because $\mathfrak{I}, H, H, \Gamma \models \mathsf{Tr}(P)$ and Theorem 5. As a consequence, $e_0.m(e_1, \ldots, e_n);$ is not stuck.
- **Return**. Suppose the sequence of statements in the top activation record is empty and that the stack contains more than 1 activation record. Since σ is valid, it follows that $\mathfrak{I}, H, G_1, \Gamma_1 \models \mathsf{vc}(\mathsf{nil}, \mathsf{Tr}(e_{post_1}) \wedge \mathsf{frame}(\mathsf{Tr}(e_{mod_1})))$. Therefore, $\mathfrak{I}, H, G_1, \Gamma_1 \models \mathsf{Tr}(e_{post_1}) \wedge \mathsf{frame}(\mathsf{Tr}(e_{mod_1}))$. The validity of π implies that m, the method currently being executed on the top of the stack, is a valid mutator. Validity of m implies that

  $$\Sigma_\pi \vdash \forall h, h_{old}, \mathit{this}, x_1, \ldots, x_n \bullet \mathit{wf}(h_{old}) \wedge \mathit{succ}(h_{old}, h) \wedge \\ \mathit{this} \neq \mathit{null} \wedge \mathsf{allocated}_{h_{old}}(\mathit{this}, x_1, \ldots, x_n) \wedge \mathsf{Tr}(\mathbf{old}(e_{pre})) \Rightarrow \mathsf{Df}(e_{post})$$

  Since $\mathfrak{I} \models \Sigma_\pi$, it follows that

  $$\mathfrak{I}, H, G_1, \Gamma_1 \models \mathit{wf}(h_{old}) \wedge \mathit{succ}(h_{old}, h) \wedge \\ \mathit{this} \neq \mathit{null} \wedge \mathsf{allocated}_{h_{old}}(\mathit{this}, x_1, \ldots, x_n) \wedge \mathsf{Tr}(\mathbf{old}(e_{pre})) \Rightarrow \mathsf{Df}(e_{post})$$

  Since σ is a valid configuration, the heap G_1 is well-formed and H is a successor of G_1. The variables $\mathbf{this}, x_1, \ldots, x_n$ are allocated in G_1 since they were part of Γ_1 at the time the activation record was put onto the call stack. Finally, the activation record second from the top is valid, hence the precondition evaluates to **true** in G_1. Since the postcondition is well-defined and $\mathfrak{I}, H, G_1, \Gamma_1 \models \mathsf{Df}(e_{post_1}) \wedge \mathsf{Tr}(e_{post_1})$, it follows that the postcondition evaluates to **true**. Via similar reasoning, we can deduce that computing the modifies clause does not get stuck and is satisfied. Therefore, the return is not stuck.
- **Object creation**. Suppose the statements in the top activation record of σ are $x := \mathbf{new}\ C;\ \overline{s}$. An object creation cannot be stuck.
- **Assert**. Suppose the statements in the top activation record of σ are $\mathbf{assert}\ e;\ \overline{s}$. Since σ is a valid configuration, it follows that $\mathfrak{I}, H, G_1, \Gamma_1 \models \mathsf{Df}(e) \wedge \mathsf{Tr}(e)$. It follows from Theorem 5 that e is not stuck and that e evaluates to **true**. Hence, $\mathbf{assert}\ e;$ is not stuck.

□

Theorem 8 (Preservation). $\rightarrow$ preserves validity of program states.

$$\forall \sigma, \sigma' \bullet \mathsf{valid}(\sigma) \wedge \sigma \rightarrow \sigma' \Rightarrow \mathsf{valid}(\sigma')$$

Proof. By case analysis on the step taken. We abbreviate the formula $\mathsf{Tr}(e_{post_i}) \wedge \mathsf{frame}(e_{mod_i})$ to ψ_i.

- **Variable declaration**. Suppose the statements in the top activation record of σ are $C\ x;\ \overline{s}$. Since σ is a valid configuration, it follows that $\mathfrak{I}, H, G_1, \Gamma_1 \models \forall x \bullet \mathsf{vc}(\overline{s}, \psi_1)$. It immediately follows that $\mathfrak{I}, H, G_1, \Gamma_1[x \mapsto \mathit{null}] \models \mathsf{vc}(\overline{s}, \psi_1)$.
- **Variable update**. Suppose the statements in the top activation record of σ are $x := e;\ \overline{s}$. Since σ is a valid configuration, it follows that $\mathfrak{I}, H, G_1, \Gamma_1 \models \mathsf{Df}(e) \wedge \mathsf{vc}(\overline{s}, \psi_1)[\mathsf{Tr}(e)/x]$. It follows from Theorem 5 that $\mathfrak{I}, H, G_1, \Gamma_1 \models \mathsf{Tr}(e) = v$, where $H, G_1, \Gamma_1 \vdash e \Downarrow v$. Therefore, $\mathfrak{I}, H, G_1, \Gamma_1[x \mapsto v] \models \mathsf{vc}(\overline{s}, \psi_1)$.
- **Field update**. Suppose the statements in the top activation record of σ are $e_1.f := e_2;\ \overline{s}$. Since σ is a valid configuration, it follows that $\mathfrak{I}, H, G_1, \Gamma_1 \models \mathsf{Df}(e_1) \wedge \mathsf{Df}(e_2)$ and $\mathfrak{I}, H, G_1, \Gamma_1 \models (\mathit{succ}(h, h[(\mathsf{Tr}(e_1), f) \mapsto \mathsf{Tr}(e_2)]) \Rightarrow \mathsf{vc}(\overline{s}, \psi_1)[h[(\mathsf{Tr}(e_1), f) \mapsto \mathsf{Tr}(e_2)]/h]$. It follows from Theorem 5 that $\mathfrak{I}, H, G_1, \Gamma_1 \models \mathsf{Tr}(e_1) = v_1 \wedge \mathsf{Tr}(e_2) = v_2$, where $H, G_1, \Gamma_1 \vdash e_1 \Downarrow v_1$ and $H, G_1, \Gamma_1 \vdash e_2 \Downarrow v_2$. Because $H[(v_1, f) \mapsto v_2]$ is a successor of H, $\mathfrak{I}, H[(v_1, f) \mapsto v_2], G_1, \Gamma_1 \models \mathsf{vc}(\overline{s}, \psi_1)$.
- **Mutator invocation**. Suppose the statements in the top activation record of σ are $e_0.m(e_1, \ldots, e_n);\ \overline{s}$. The activation record second from the top in σ' is trivially valid. Why is the new top activation record valid? Since σ is a valid configuration, it follows that $\mathfrak{I}, H, G_1, \Gamma_1 \models \mathsf{Df}(e_0) \wedge \ldots \wedge \mathsf{Df}(e_n) \wedge \mathsf{Tr}(P)$, where P is $\mathsf{mpre}(C, m)[\mathsf{Tr}(e_0)/\mathbf{this}, \ldots, \mathsf{Tr}(e_n)/x_n]$. It follows from Theorem 5 that for each $i \in \{0, \ldots, n\}$, $J, H, G_1, \Gamma_1 \models \mathsf{Tr}(e_i) = v_i$ where $H, G_1, \Gamma_1 \vdash e_i \Downarrow v_i$.
  Since π is a valid program, m is a valid mutator. Validity of m implies that

$$\begin{array}{l} \Sigma_\pi \vdash \forall h, \mathit{this}, x_1, \ldots, x_n \bullet \mathit{wf}(h) \wedge \mathit{this} \neq \mathit{null} \wedge \\ \qquad \mathsf{allocated}_h(\mathit{this}, x_1, \ldots, x_n) \wedge \mathsf{Tr}(e_{pre}) \\ \qquad\qquad \Downarrow \\ \qquad \mathsf{vc}(\overline{s}, \mathsf{Tr}(e_{post}) \wedge \mathsf{frame}(e_{mod}))[h/h_{old}] \end{array}$$

  Because $\mathfrak{I} \models \Sigma_\pi$, the heap H is well-formed, $v_0, \ldots, v_n$ are allocated in H and P holds, it follows that

$$\mathfrak{I}, H, H, [\mathbf{this} \mapsto v_0, \ldots, x_n \mapsto v_n] \models \mathsf{vc}(\overline{s}, \mathsf{Tr}(e_{post}) \wedge \mathsf{frame}(e_{mod}))$$

  Hence, σ' is still valid.
- **Return**. Suppose the sequence of statements in the top activation record is empty. Since the activation record second from the top is valid, it follows that

$$\begin{array}{l} \mathfrak{I}, G_1, G_2, \Gamma_2 \models (\forall h' \bullet \\ \quad \mathit{succ}(h, h') \wedge \\ \quad \mathsf{Tr}(Q)[h'/h, h/h_{old}] \wedge \\ \quad \mathsf{frame}(M)[h'/h, h/h_{old}] \\ \quad \Rightarrow \\ \quad \mathsf{vc}(\overline{s_2}, \psi_2)[h'/h] \\ ) \end{array}$$

  Let's instantiate h' with H. H is a successor of G_1. If follows from the validity of the top activation record that the postcondition holds and the modifies clause is satisfied. Therefore, $\mathfrak{I}, H, G_2, \Gamma_2 \models \mathsf{vc}(\overline{s_2}, \psi_2)$.
- **Assert**. Suppose the statements in the top activation record of σ are **assert** $e;\ \overline{s}$. Since σ is a valid configuration, it follows that $\mathfrak{I}, H, G_1, \Gamma_1 \models \mathsf{vc}(\overline{s}, \psi_1)$.

□

Theorem 9 (Soundness). Valid programs do not get stuck.

Proof. Follows from the fact that the initial state is valid and Theorems 7 and 8. □

6. Inheritance

Inheritance is a key component of the object-oriented paradigm that allows a class to be defined as an extension of one or more existing classes. For example, consider the class *BackupCell* from Figure 15. *BackupCell*

```
class BackupCell extends Cell {
  int backup;

  BackupCell()
    modifies ∅;
    ensures valid();
    ensures getX() = 0;
  { super(); backup := 0; }

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

  void undo()
    requires valid();
    modifies footprint();
    ensures valid();
    ensures getX() = old(getBackup());
    ensures fresh(footprint() \ old(footprint()));
  { super.setX(backup); }

  pure bool valid()
    reads valid() ? footprint() : *;
  { return super.valid() ∧ ({ backup } ∩ super.footprint() = ∅); }

  pure set footprint()
    requires valid();
    reads footprint();
  { return { backup } ∪ super.footprint(); }

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

Fig. 15. *BackupCell*, a subclass of *Cell*.

is an extension of its superclass *Cell*. More specifically, a *BackupCell* satisfies all properties of *Cell*, but can additionally undo the last call to *setX*. Reasoning about inheritance in verification is challenging, since the code to be executed for a call is not determined at compile-time but at run-time (depending on the type of the receiver). In particular, one must ensure that the contract used when verifying the call and the contract used when verifying the callee match. In this section, we extend the approach described in the previous sections to inheritance. The encoding of inheritance described here is similar to earlier proposals by Jacobs *et al.* [JP07], Leavens *et al.* [Lea06], Dovland *et al.* [DJOS08] and Parkinson *et al.* [PB08].

Method calls can both be statically and dynamically bound, depending on the method itself and the calling context. For example, the call to *getX* in the client code of Figure 3(b) is dynamically bound, while the call to *getX* in the body of *BackupCell.setX* is statically bound. In Java, constructors, private methods and super calls are statically bound, while all other calls are dynamically bound. To distinguish statically bound calls of pure methods from dynamically bound ones, we introduce an additional function symbol in

```
class C {
  C()
    ensures valid();
  { }

  void m(int x)
    requires valid() ∧ 0 ≤ x;
  { assert 0 ≤ x; }

  pure bool valid()
    reads ∅;
  { return true; }
}
```

(a)

```
class D extends C {
  D()
    ensures true;
  { super(); }

  void m(int x)
    requires x = 0;
  { assert x = 0; }

  pure bool valid()
    reads ∅;
  { return false; }
}
```

(b)

Fig. 16. A class C and its subclass D.

the verification logic. More specifically, for a pure method p defined in a class C, the signature not only includes a symbol $C.p$ but additionally contains a function symbol $C.p_D$. The former symbol is used to encode 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 equals C.

$$\begin{array}{c} \forall h, this, x_1, \ldots, x_n \bullet \\ type(this) = C \\ \Downarrow \\ C.p(h, this, x_1, \ldots, x_n) = C.p_D(h, this, x_1, \ldots, x_n) \end{array}$$

Note that *type* is a function that returns the dynamic type of an object. Furthermore, whenever a method $D.p$ overrides $C.p$, we add the following axiom to our theory:

$$\begin{array}{c} \forall h, this, x_1, \ldots, x_n \bullet \\ type(this) <: D \\ \Downarrow \\ C.p_D(h, this, x_1, \ldots, x_n) = D.p_D(h, this, x_1, \ldots, x_n) \end{array}$$

The above axiom encodes the property that dynamically bound calls to $C.p$ and $D.p$ are equal if the receiver is a subtype (denoted $<:$) of D.

Invocations of pure methods with receiver **this** are treated differently in contracts and code. In normal code, such calls follow the standard rules that guide binding. However, if the **this**-call appears in the method contract of a statically bound call of a method m, then it is treated as statically bound; otherwise the **this**-call is dynamically bound. Method implementations are verified under the assumption that the corresponding method is called statically. This assumption is sound provided each subclass overrides each instance method of its superclass. Indeed, if an actual call is statically bound, then the contract seen by the caller and the contract for verifying the implementation are equal. If the actual call is dynamically bound, then the dynamic type of the receiver equals the static type of **this** for the callee and therefore the static contract equals the dynamic one (follows from the first inheritance axiom). In Figure 15, *BackupCell* overrides each method except *getX*. If a class does not override a method m, we generate a default method override which simply calls the superclass and inherits the superclass contract as is. This method is subject to verification like other methods.

Note that *BackupCell* does not depend on internal details of its superclass. As a consequence, changing *Cell*'s internal representation (within the boundaries set by its method contracts) can never break *BackupCell*.

To ensure that the implementation of a subclass D does not break the method contracts defined in a superclass C, we check that the contract of each method in C is satisfied by a method body that calls the method defined in D. More specifically, for each method m in C, we check that a method body that calls

program	time taken	source
Cell	0.4	[PB05, JP07, DP08]
Interval	1.4	[]
ArrayList, Stack and Iterator	4.8	[Kas06b, Kas06a, Par05, LS07]
SubjectObserver (1 observer)	1	[Par07, BNR08, LS07]
BackupCell	0.6	[PB08, JP07]
MasterClock	6.2	[BN04]

Fig. 17. Table showing the time taken (in seconds) to verify each program.

$D.m$ 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 can be broken by the addition of a new subclass.

The rules for subclassing outlined above generalize Liskov's substitution principle [LW94]. For example, consider the classes C and D from Figure 16. D is a valid subclass of C, even though $D.m$'s precondition is stronger than the precondition of its overridden method, $C.m$. This kind of subclassing is safe, since it is not possible to establish the precondition of a dynamically bound call $o.m()$ where o's static type is C and its dynamic type is D. Indeed, $o.valid()$ never holds for an object with dynamic type D. That is, one can never pass an object with dynamic type D to a method that expects a valid instance of C.

The extension for dealing with inheritance described above (which is largely based on [PB08]) solves the extended state problem [Lei98, Mül02] by allowing predicates and pure methods to be redefined in subclasses. For example, the predicate *BackupCell.valid* overrides and redefines *Cell.valid* to account for the additional field, *backup*. The meaning of $o.valid()$ is determined by the dynamic type of o. For instance, if o's dynamic type is *Cell*, then one should interpret $o.valid()$ as defined in the class *Cell*. However, if the dynamic type of o is unknown, then precise meaning of $o.valid()$ cannot be determined. As consequence, $o.valid()$ does not in general imply that $o.x$ can be updated, as redefinitions of *valid* do not have to include the permission to access $o.x$. For example, the following code snippet does not verify.

$$\begin{array}{l}\textbf{void } assignToX(Cell\ o)\\ \quad \textbf{requires } o \neq null \wedge o.valid();\\ \{\ o.x := 1;\ \}\end{array}$$

Note that the assignment in the method *setX* of Figure 15 does verify, because we assume **this**-calls in method contracts are statically bound when verifying method implementations. As argued above, this assumption is sound only if each virtual method is overridden and reverified in each subclass.

7. Experience

To show the verification approach described in the previous section is amenable to automatic static verification, we implemented it in a verifier prototype. The prototype was used to verify several challenging examples from related work. The experiments were executed on a desktop machine running Windows Vista with a Pentium Core duo 2.66 GHz processor and 4 GB of memory. To discharge the verification conditions, we used Z3 [dMB08], a satisfiability-modulo-theory (SMT) solver. The verifier itself and the programs shown in Table 17 can be downloaded from `http://www.cs.kuleuven.be/~jans/DFJ`.

The remainder of this section discusses a number of important extensions used in the implementation to make the approach described above practical.

7.1. Modularity

It is sound to use the implementation axioms of a library during verification of client code. For example, it is ok to rely on *getX*'s implementation axiom when verifying Figure 3(b). However, when the correctness proof of code relies on implementation axioms of methods implemented in other modules, that code's correctness becomes dependent on another module's internal details. For that reason, our verifier prototype makes a pure method's implementation axiom available only to other methods implemented in the same module. Indeed, to prove the correctness of Figure 3(b), it suffices to rely on *Cell*'s frame and footprint allocated axioms.

7.2. Triggers

An important consideration in the design of a verifier that relies on an SMT solver to discharge verification conditions is what triggers to use. A trigger consists of one or more terms. The SMT solver uses triggers to determine when and how it should instantiate universal quantifiers. Z3 requires a trigger to mention all quantified variables.

The implementation, frame and allocated axioms all contain a universal quantifier. The trigger for the implementation axiom is $C.p(h, this, x_1, \ldots, x_n)$. The trigger for the allocated axiom is the term $(o, f) \in C.p(h, this, x_1, \ldots, x_n)$. Finally, the trigger for the frame axiom consists of the terms $succ(h_{old}, h)$ and $C.p(h, this, x_1, \ldots, x_n)$. In an earlier version of the tool, we instead used $wf(h_{old})$, $wf(h)$ and $C.p(h, this, x_1, \ldots, x_n)$ as a trigger for the frame axiom. However, this choice was considerably slower as the theorem prover had to consider $n \times (n-1)$ possible instantiations (any two well-formed heaps) instead of $n-1$ (any two successor heaps), where n is the number of heaps appearing in the program text.

7.3. Defaults

Parkinson [Par07] argues that program verifiers should not include special, built-in rules for reasoning about object invariants, since they can be encoded as predicates in the underlying logic and because existing approaches with built-in rules for invariants cannot handle certain programming idioms such as the Subject-Observer pattern. The latter approach is also adopted by this paper. Contrary to Parkinson, Summers *et al.* [SDM09] argue that object invariants lie at the heart of object-oriented programming, and that they must therefore receive native support from a verification technique. We believe a program verifier can and should provide defaults and syntactic sugar for object invariants. For example, a verifier could support invariant annotations on classes, which are desugared during verification into boolean pure methods and additional pre- and postconditions on the methods of the corresponding classes. If the defaults offered by the verifier are too stringent, then the developer should be able to disable them for particular parts of the code base and be able to fall back on using the underlying machinery.

A possible defaulting scheme that covers many typical scenarios is the following. Instead of invariant methods, developers can write invariant annotations on classes. An invariant annotation consists of the keyword **invariant** followed by a boolean expression. The tool desugars this annotation into an invariant method that returns this boolean expression conjoined with a number of invariant method calls and disjointness conditions. The latter invariant method calls and disjointness conditions are inferred from the fields of the class and from annotations on those fields. More specifically, for each field f marked **rep**[3] the generated invariant method includes a call to the invariant method of the object referred to by f. In addition, for each field, a disjointness condition is added which states that the location of that field is not included in a footprint of one of the objects referred to by a rep-field. Finally, the restriction that the footprints of all rep-objects are mutually disjoint is added to the invariant method. An invariant declaration also gives rise to a number of pre- and postconditions. More specifically, a precondition requiring the generated invariant method to return true is automatically added to each non-constructor method and a similar postcondition is added to each constructor and mutator. For each class, the tool automatically generates a dynamic frame based on the fields of that class. The body of the dynamic frame contains a location for each field of the class. Moreover, if the field is marked **rep**, then the footprint of the object referred to by that field is also included in the generated footprint method. The modifies clause of a mutator and the reads clause of a pure method default to a call of the footprint method. In addition, for each mutator an additional postcondition is generated which states that the footprint is only extended with fresh locations.

The defaulting scheme described above is applied to the classes *ArrayList* and *Stack* in Figure 18. Desugaring the defaults leads to the classes shown in Figures 5 and 6. Figure 18 contains 13 lines of annotations[4] (2 rep modifiers, 2 invariant declarations, 6 postconditions and 3 pure modifiers), while Figures 5 and 6 contain 33 such lines (7 preconditions, 8 postconditions, 4 modifies clauses, 7 reads clauses and 7 pure mod-

[3] The **rep** annotation on fields is taken from Spec# [BLS04] where it denotes that the representation of the object containing the field includes the object pointed to by the field (i.e. the former object owns the latter).

[4] The number of annotated lines includes lines with rep and pure annotations, invariant declarations and method contracts. The bodies of pure methods are not considered as annotated lines.

```
class ArrayList {
  rep Object[] items;
  int size;

  invariant items ≠ null ∧ 0 ≤ size ≤ items.length;

  ArrayList()
    ensures size() = 0;
  { items := new Object[5]; size := 0; }

  void add(Object o)
    ensures size() = old(size()) + 1;
    ensures get(size() − 1) = o;
    ensures ∀i ∈ (0 : size() − 1) • get(i) = old(get(i));
  { ... }

  pure int size()
  { return size; }

  pure Object get(int index)
  { return items[index]; }
}
```

(a)

```
class Stack {
  rep ArrayList contents;

  invariant contents! = null;

  Stack()
    ensures size() = 0;
  { contents := new ArrayList(); }

  void push(Object o)
    ensures size() = old(size() + 1);
  { contents.add(o); }

  pure int size()
  { return contents.size(); }
}
```

(b)

Fig. 18. The classes *ArrayList* and *Stack* with defaults.

ifiers). A further reduction in the number of annotations could be achieved by using non-null types as in Spec# [BLS04].

In general, the defaulting scheme ensures that (1) the validity of the receiver must not be mentioned explicitly in method contracts, (2) that reads and modifies clauses can be omitted as they default to the generated dynamic frame, (3) that dynamic frames and invariants are (at least partially) generated based on the fields of the class and (4) that postconditions which ensure that only fresh locations are added to footprints default to freshness of objects added to the dynamic frame.

7.4. Flexible Pure Methods

It is crucial for soundness that the axioms generated for pure methods are consistent if the program is valid. For example, consider the pure method *bad* shown below.

```
pure int bad() { return bad() + 1; }
```

bad's implementation axiom is inconsistent: there is no interpretation for the function *bad* such that $bad() = bad() + 1$. To ensure verification is sound, we must somehow detect such "dangerous" pure methods. In this paper, we ensure consistency by enforcing a simple, but very restrictive rule: a method can only call pure methods defined earlier in the program text. This rule guarantees that pure methods terminate, which in turn implies that their implementation axioms are consistent. However, this rule is overly restrictive as recursive specifications of linked data structures would not be well-formed.

In our implementation, we instead use more flexible rules. More specifically, the body of a normal pure method p_2 can call any other pure method p_1 if either p_1 is defined before p_2 or if the size of p_1's reads clause is strictly smaller than p_2's reads clause. In addition, a pure method can be annotated with a measure (similar to JML's measured_by annotations [LPC+07]). The pure method can call itself if its measure decreases. Ensuring consistency of the encoding of pure methods is an active area of research [LM09, RADM08].

7.5. Pure Methods Postconditions

In the specification language of Figure 9, it is impossible to express and impose constraints on the return values of pure methods. However, the correctness of client code often relies on the fact that return values satisfy certain properties. For example, users of the class *ArrayList* might depend on the property that *getSize* never returns a negative number.

Our implementation allows developers to express constraints on the return values of pure methods via postconditions on those methods. Our tool exposes such postconditions to client code via axioms. More specifically, for each pure method

$$\begin{array}{l}\textbf{pure } t\ p(C_1\ x_1, \ldots, C_n\ x_n)\\ \quad \textbf{requires } e_{pre};\ \textbf{reads } e_{read};\ \textbf{ensures } e_{post};\\ \{\ \textbf{return } e_{body};\ \}\end{array}$$

defined in a class C, we generate a postcondition axiom

$$\begin{array}{c}\forall h, this, x_1, \ldots, x_n \bullet\\ wf(h) \wedge this \neq null \wedge \mathsf{allocated}_h(this, x_1, \ldots, x_n) \wedge \mathsf{Tr}(e_{pre})\\ \Downarrow\\ \mathsf{Tr}(e_{post})[C.p(h, this, x_1, \ldots, x_n)/result]\end{array}$$

Informally, this axiom says that the postcondition holds, whenever the precondition is true. To ensure that generated axiom is consistent, termination of pure methods must be enforced.

7.6. Limitations of the Tool

The prototype implementation has a number of limitations. First of all, the tool models Java integers as mathematical numbers and does not take overflow into account. This can lead to unsoundness, as the assumptions that the verifier makes are incorrect in the presence of overflow. Secondly, the prototype relies on an SMT solver to discharge the generated verification conditions. The current generation of SMT solvers however does not construct inductive proofs. Induction is often necessary when reasoning about recursive pure methods and linked data structures. This limitation can be addressed by using lemma methods [JP08] to encode inductive proofs. Thirdly, the time needed for discharging the verification conditions increases as the number of statements that modify the heap increases. We believe this increase is mainly caused by the fact that after each heap modification a large number of frame axioms must be applied to reconstruct the information that is known about the state. Recent advances in symbolic execution [DP08, BCO05, JP08] show that verification times can dramatically be reduced. We believe a verification approach for dynamic frames based on symbolic execution would be very interesting and promising line of future research. Finally, the verifier prototype can currently not be applied to real Java programs as many features offered by the Java programming language, such as exceptions, increment operators, break statements and for loops, are not supported.

8. Related Work

The approach presented in this paper was inspired by the work of Kassios [Kas06b, Kas06a]. Kassios uses specification variables, similar to our pure methods, to achieve data abstraction. To solve the framing problem, he proposes using dynamic frames to abstractly specify the footprint of specification variables and the effect of mutator methods. Dynamic frames are specification variables that hold sets of memory locations. However, Kassios' solution is presented in the context of an idealized logical framework. We show how Kassios' ideas can be incorporated in a program verifier for a Java-like language based on first-order logic, and demonstrate that many interesting examples can be verified automatically.

Banerjee *et al.* [BNR08, BBN08] and Leino [Lei08] both propose similar techniques for automating the ideas of dynamic frames. Contrary to the approach proposed in this paper, they rely on ghost fields[5] instead of pure methods to represent footprints. As an example, consider the class *Cell* of Figure 19. The field

[5] A ghost field is a field that is only used for verification. It can be ignored during a program's execution.

```
class Cell {
  int x;
  ghost set footprint;

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

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

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

  pure boolean valid()
    reads footprint;
  { return { x, footprint } ∈ footprint; }
}
```

Fig. 19. The class *Cell* specified in regional logic.

footprint is a ghost field of type **set**. This set contains the locations that represent the corresponding *Cell* object. *Cell*'s methods specify what they read or write in terms of this ghost field. The invariant constrains the field to ensure that it contains the necessary locations. If one would omit this constraint in the invariant, verification of *setX* and *getX* would fail, since *footprint* might not contain the location **this**.x. An advantage of using ghost fields instead of pure methods is that recursive data structures can be specified by quantifying over the elements of the footprint in the invariant (instead of using recursive pure methods). Because ghost fields are updated explicitly, the theorem prover does not have to "compute" the new value of footprint fields after a state update. A disadvantage of using ghost fields is that they must be initialized and updated explicitly by the programmer. For example, *Cell*'s constructor initializes *footprint* to the set { x, *footprint* }. In general, any mutator that changes the set of locations that make up the footprint must also update the ghost field.

The implicit dynamic frames approach [SJP08, SJP09] is a variant of the traditional dynamic frames approach where frame information is inferred from method preconditions. Both approaches have similar expressive power. An advantage of the traditional approach is that read effects for mutators need not be declared in the method contract. However, we believe that the implicit dynamic frames approach subsumes the approach described in this paper when verifying concurrent programs, because read accesses must be checked for avoiding data races in concurrent programs.

Parkinson and Bierman [Par05, PB05, PB08] extend separation logic [Rey02] to the Java programming language, introducing abstract predicates to attain data abstraction and predicate families to deal with inheritance. One difference between their approach and ours is that we allow heap-dependent expressions (field accesses and pure method invocations) to be used inside method contracts. This style of annotating code may be more familiar to programmers, as they do not have to learn a new specification language. They recently implemented their approach in a tool, called jStar [DP08]. jStar relies on symbolic execution, while we use the more traditional combination of verification condition generation and automated theorem proving.

In the basic Boogie methodology [BDF+04], data abstraction is limited to object invariants. More specifically, each object has a ghost field *inv*, and the methodology ensures that the invariant of an object o

holds whenever $o.inv$ is true. To ensure the soundness of the approach, the Boogie methodology imposes several restrictions: *inv* can only be updated using special operations called **pack** and **unpack**, updating a field $o.f$ requires $o.inv$ to be false, and finally the invariant itself can only mention fields within the object's ownership cone. The dynamic frames approach can be considered to be conceptually simpler than the Boogie methodology (and its extensions), since it does not impose any methodological restrictions.

Leino and Müller [LM04] and Barnett and Naumann [BN04] extend the basic Boogie methodology to deal with non-hierarchical object structures. In particular, they allow invariants to mention fields outside of the object's ownership cone provided certain visibility requirements are met. More specifically, if the invariant of class C mentions a field f of a non-owned object, then C must be visible in the the class declaring f. No such restriction is present in our approach.

Jacobs and Piessens [JP07] and Darvas and Leino [DL07] both extend the basic Boogie methodology with support for data abstraction using pure methods. Similarly to our approach, they model pure methods as functions in the verification logic. Both approaches essentially solve the framing problem by encoding in the verification logic that these functions depend only on a number of ownership cones instead of on the entire heap.

Leino and Müller [LM06] extend the basic Boogie methodology with model fields to achieve data abstraction. A model field declaration consists of a type, a name, and a constraint. A model field cannot be assigned to within the program text; instead the model field is assigned a random value satisfying the constraint whenever the object is being packed. To prevent unsound reasoning arising from unsatisfiable constraints, Leino and Müller require the theorem prover to come up with a witness before assuming the constraint holds. However, experience shows that theorem provers (in particular Simplify) are unable to find witnesses even in simple cases, and as such it is unlikely that their approach is suitable for use within an automatic program verifier.

In [LPHZ02], 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 [LPHZ02] cannot. For example, the former restriction rules out sharing of representation objects, as is the case in the iterator pattern.

Müller's thesis [Mül02] combines model fields with an ownership type system called Universes. Model fields are similar to pure methods that have no parameters. Model fields may depend on the fields of owned objects and the fields of peer objects, i.e. objects with the same owner as the receiver. However, model fields can only depend on peers if a model field is visible within the peer. For example, if the pure method *hasNext* from Figure 7 were a model field, then *hasNext* would have to be visible to the class ArrayList. Our approach has no such restriction.

The use of pure methods in specifications has been discussed extensively in the literature [RADM08, DM06, JP07, BNSS04, DL07]. In particular, encoding pure methods as functions in the logic is a standard technique in verification.

9. Conclusion

We demonstrate that the dynamic frames approach can be integrated into an automatic verifier based on verification condition generation and automated theorem proving. The approach has been proven sound and it has been implemented in a verifier prototype. The verifier has been used to prove correctness of several programming patterns considered challenging in related work.

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. The authors would like to thank the anonymous reviewers for their feedback and suggestions.

References

[BBN08] Anindya Banerjee, Mike Barnett, and David A. Naumann. Boogie meets regions: a verification experience report. In *VSTTE*, 2008.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.

[BDF+04] Mike Barnett, Robert DeLine, Manuel Fahndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6), 2004.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS*, 2004.

[BMR95] Alex Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering (TSE)*, 21(10), 1995.

[BN04] Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Mathematics of Program Construction*, 2004.

[BNR08] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *European Conference on Object-oriented Programming (ECOOP)*, 2008.

[BNSS04] Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specifications. In *Formal Techniques for Java-like Programs (FTFJP)*, 2004.

[BPP03] Gavin Bierman, Matthew Parkinson, and Andrew Pitts. An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2003.

[DE97] Sophia Drossopoulou and Susan Eisenbach. The Java type system is sound – probably. In *European Conference on Object-oriented Programming (ECOOP)*, 1997.

[DJOS08] Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen. Lazy behavioral subtyping. In *Formal Methods (FM)*, volume 5014, 2008.

[DL07] Ádám Darvas and K. Rustan M. Leino. Practical reasoning about invocations and implementations of pure methods. In *Fundamental Approaches to Software Engineering (FASE)*, 2007.

[DM06] Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology (JOT)*, 5(5), 2006.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[DP08] Dino Distefano and Matthew Parkinson. jStar: Towards practical verification for Java. In *Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2008.

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1999.

[JP07] Bart Jacobs and Frank Piessens. Inspector methods for state abstraction. *Journal of Object Technology (JOT)*, 6(5), 2007.

[JP08] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report 520, Department of Computer Science, Katholieke Universiteit Leuven, 2008.

[Kas06a] Yannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Formal Methods (FM)*, 2006.

[Kas06b] Yannis T. Kassios. *A Theory of Object Oriented Refinement.* PhD thesis, University of Toronto, 2006.

[Lea06] Gary T. Leavens. JML's rich, inherited specifications for behavioral subtypes. In *International Conference on Formal Engineering Methods (ICFEM)*, 2006.

[Lei98] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1998.

[Lei08] K. Rustan M. Leino. Specification and verification of object-oriented software. *Marktoberdorf International Summer School – Lecture Notes*, 2008.

[LLM07] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing (FACS)*, 19(2), 2007.

[LM04] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *European Conference on Object-oriented Programming (ECOOP)*, 2004.

[LM06] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In *European Symposium on Programming (ESOP)*, 2006.

[LM09] K. Rustan M. Leino and Ronald Middelkoop. Proving consistency of pure methods and model fields. In *Fundamental Approaches to Software Engineering (FASE)*, 2009.

[LN02] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(5), 2002.

[LPC+07] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. JML reference manual, 2007.

[LPHZ02] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[LS07] K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. In *European Symposium on Programming (ESOP)*, 2007.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6), 1994.

[LW98] Gary T. Leavens and Jeannette M. Wing. Protective interface specifications. *Formal Aspects of Computing (FACS)*, 10(1), 1998.

[Mül02] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[Par05] Matthew Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.

[Par07] Matthew Parkinson. Class invariants: The end of the road? In *International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming (IWACO)*, 2007.

[PB05] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Principles of Programming Languages (POPL)*, 2005.

[PB08] Matthew Parkinson and Gavin Bierman. Separation logic, abstraction and inheritance. In *Principles of Programming Languages (POPL)*, 2008.

[RADM08] Arsenii Rudich, Ádám Darvas, and Peter Müller. Checking well-formedness of pure-method specifications. In *Formal Methods (FM)*, 2008.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logic in Computer Science (LICS)*, 2002.

[SDM09] Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. The need for flexible object invariants. In *International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming (IWACO)*, 2009.

[SJP08] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. In *Formal Techniques for Java-like Programs (FTFJP)*, 2008.

[SJP09] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-oriented Programming (ECOOP)*, 2009.

[SJPS08] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *Fundamental Approaches to Software Engineering (FASE)*, 2008.