An Automatic Verifier for Java-like Programs Based on Dynamic Frames

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

[1] Katholieke Universiteit Leuven, Belgium
{jans,bartj,frank}@cs.kuleuven.be
[2] Microsoft Research Redmond, USA
schulte@microsoft.com

Abstract. 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. In object-oriented programs, classes typically enforce data abstraction by providing access to their internal state only through methods. By using method calls in method contracts, data abstraction can be extended to specifications. In this paper, methods used for this purpose must be side-effect free, and are called pure methods.
We present an approach to the automatic verification of object-oriented programs that use pure methods for data abstraction. The cornerstone of our approach is the solution to the framing problem, i.e. client code must be able to determine whether state changes affect the return values of pure methods. More specifically, we extend each method contract with a method footprint, an upper bound on the memory locations read or written by the corresponding method. Footprints are specified using dynamic frames, special pure methods that return sets of memory locations. Thanks to this abstraction, implementations can evolve independently from specifications, loosely coupled only by pure functions.
We implemented this approach in a custom build of the Spec# program verifier, and used it to automatically verify several challenging programs, including the iterator and observer patterns. The verifier itself and the examples shown in this paper can be downloaded from the authors' homepage [1].

1 Introduction

The principle of data abstraction is a central concept in object-oriented programming. That is, a class typically hides its implementation details from clients, and instead offers methods to access its internal state. Adherence to the principle of data abstraction ensures that client code remains independent of the implementation of classes it is using, and as a consequence that changing the implementation of a class (within the boundaries described by its contract) does not affect clients. For example, consider the class *Cell* shown in Figure 1. Each *Cell* object holds an integer value which is stored in the private field x. Client code can only access x through the getter *getX* and the setter *setX*. Since clients only depend on the getter and setter, changing *Cell*'s internal representation does not affect them. In particular, changing the implementation will not affect the correctness of the client program of Figure 1(b).

To preserve data abstraction within specifications, specifications must be written in an implementation-independent manner. In particular, specifications should not expose the private fields of a class. One way to achieve this independence is to use method calls within specifications. In this paper, methods used for this purpose must be side-effect free, and are called pure methods. Non-pure methods are called mutators. For example, the behavior of the mutator *setX* is specified by describing its effect on the pure method *getX*. The specification of *Cell* never mentions the field x. Using pure methods within specifications

```
class Cell {
  private int x;

  Cell()
    writes ∅;
    ensures getX() = 0;
    ensures footprint().isFresh();
  { }

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

  void setX(int value)
    writes footprint();
    ensures getX() = value;
    ensures footprint().newElemsFresh();
  { x := value; }

  pure set footprint()
    reads footprint();
  { return { &x }; }
}
```

(a)

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

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

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

(b)

Fig. 1. A class *Cell* and a client program.

gives rise to a framing problem [2, Challenge 3], i.e. client code must be able to determine the effect of heap changes on the return values of pure methods. For instance, to show that $c_1.getX()$ equals 5 at the end of the code snippet in Figure 1(b), we must be able to deduce that creating and modifying c_2 does not affect the state of c_1. This deduction should not rely on *Cell*'s implementation, since doing so would break information hiding.

Recently, Kassios [3, 4] proposed a promising solution to the framing problem. More specifically, he proposes using dynamic frames, specification variables (similar to pure methods) that return sets of memory locations, to specify the effect of mutators and the dependencies of specification variables in an abstract manner. However, his solution is formulated in the context of an idealized, higher-order logical framework. For example, it does not show how to apply the approach to Java-like inheritance. Furthermore, the proposed approach is not applied in the context of an automatic program verifier for first-order logic.

In this paper, we propose an approach to the automatic verification of annotated Java-like object-oriented programs that combines pure methods to achieve data abstraction with Kassios' solution to the framing problem. More specifically, to solve the framing problem, we extend each method contract with a method footprint which specifies an upper bound on the memory locations read or written by the corresponding method. A memory location is an (object identifier, field name) pair. The footprint of a pure method (**reads** annotation) specifies an upper bound on the memory locations the pure method depends on, while a mutator's footprint (**writes** annotation) specifies an upper bound on the locations writable by the method. To prove that a heap change (i.e. a field update or mutator invocation) does not affect the return value of a pure method, one simply has to show that the footprint of the state change is disjoint from the pure method's footprint.

In our running example, the method footprint of both *getX* and *setX* is the singleton containing the receiver's field x. However, saying so explicitly in the method contract would expose the field x to clients and break information hiding. To specify method footprints in an implementation-independent manner, we allow developers to define dynamic frames, special pure methods that return a set of memory locations. These dynamic frames can then be used to abstractly specify method footprints. The method *footprint* is an example of a dynamic frame, and it is used to specify the footprint of all of *Cell*'s methods.

Given *Cell*'s specification, we can now prove the assertion at the end of the code snippet in Figure 1(b). Informally, the reasoning goes as follows. The specification of *Cell* guarantees that the constructor only writes to locations that were unallocated in the method pre-state, and that the new object's footprint contains only such locations. Since footprints of existing objects contain only allocated locations, the assignment $c_1 :=$ **new** *Cell*(); creates a new object whose footprint is disjoint from any existing object's footprint. *setX*'s postcondition ensures that $c_1.getX()$ equals 5 after the call statement $c_1.setX(5)$;. Furthermore, the mutator's specification ensures that it only modifies unallocated locations and locations in receiver's pre-state footprint, and that it only adds newly allocated objects to that footprint. The next assignment $c_2 :=$ **new** *Cell*(); creates a new footprint for c_2 disjoint from any other footprint. Because of this disjointness, the following statement $c_2.setX(10)$; affects neither $c_1.getX()$ nor $c_1.footprint()$. It follows that the assertion $c_1.getX()$ equals 5 *still* holds despite the intervening creation of and update to c_2.

In summary, the *contributions* of this paper are the following:

– We propose an approach to the automatic verification of Java-like object-oriented programs that combines the use of pure methods for data abstraction with Kassios' approach to solve the framing problem. In particular, we show how Kassios' solution applies to Java-like inheritance [3, Future Work].
– We implemented our approach in a tool [1], and used it to automatically verify challenging examples, such as the iterator and observer patterns.

The remainder of this paper is structured as follows. In Section 2, we explain how programs such as the one shown in Figure 1 can be verified automatically. In Section 3, we demonstrate the expressive power of our approach by showing how it verifies various object-oriented programming and specification patterns. Section 4 extends the solution of Section 2 with support for inheritance. Finally, we discuss our experience with the verifier prototype, compare with related work and conclude in Sections 5, 6 and 7.

2 Solution

SJava In this paper we restrict our attention to a small Java-like language, called SJava. SJava does not include features such as exceptions and multi-threading. However, SJava extends regular Java in three ways:

(1) SJava introduces a new primitive type **set**. An expression of type **set** represents a set of memory locations. A memory location is an (object reference, field name) pair, and the location corresponding to $e.f$ is denoted by $\&e.f$. The standard mathematical set operations such as $\cup$, $\cap$, and $\in$ can applied to expressions of type set. In addition, postconditions can apply *isFresh* and *newElemsFresh* to expressions of type set: $s.\mathit{isFresh}()$ is a two-state predicate expressing that s contains only fresh locations (i.e. locations corresponding to objects that were not allocated in the method pre-state), and $s.\mathit{newElemsFresh}()$ is a two-state predicate denoting that only fresh locations are added to s. *universe* denotes the set of all locations. $\mathbf{elems}(a)$ denotes the locations corresponding to the elements of the array a.

(2) A method in an SJava program can be marked with a **pure** annotation, indicating that it can be used in specifications. The body of a pure method consists of a single return statement returning a side-effect free expression. An expression is side-effect free if it does not contain object or array creations, simple or compound assignments, increment or decrement operators, and only calls pure methods. Non-pure methods are called *mutators*. Pure methods with return type **set** are called *dynamic frames*.

(3) Each method has a corresponding method contract, consisting of preconditions, a method footprint and postconditions. Preconditions and postconditions are boolean side-effect free expressions. The former define valid method pre-states, while the latter define valid method post-states. A method footprint is a side-effect free expression of type **set**. The footprint of a pure method (**reads** annotation) specifies the locations that can potentially be be read by the method, while a mutator's footprint (**writes** annotation) specifies the locations that can be modified by the method. More specifically, a mutator can only modify $o.f$

if $\&o.f$ is in the method's footprint or if o was unallocated at the start of the method. Pure methods have no need for writes clauses, since by definition they are not allowed to modify any location. Mutators have no need for reads clauses, and can read any location. Indeed, the effect of heap changes on the return values of mutators is not relevant in our approach (no axiom is generated to frame the return value of mutators) as only pure methods can be used in specifications. Only parameters and the variable **this** may occur free in preconditions and footprints. Postconditions can additionally mention the variable **result**, denoting the return value of the method. Furthermore, postconditions may contain old expressions **old**(e), denoting the value of the expression e in the method's pre-state.

In this section, we consider only SJava without inheritance. Section 4 explains how inheritance can be supported.

Verification Our verifier takes an SJava program as input and generates, via a translation into an intermediate verification language, a set of verification conditions. The verification conditions are first-order logical formulas whose validity implies the correctness of the program. The formulas are analyzed automatically by satisfiability-modulo-theory (SMT) solvers. Our approach is based on a general approach described in [5]. In this subsection, we focus on novel aspects of our approach: namely the way pure methods and their contracts are modeled in the verification logic and the way method footprints are enforced.

Notation Heaps are modeled in the verification logic as maps from object references and field names to values. For example, the expression $h[o, f]$ denotes the value of the field f of object o in heap h. The function *wf* returns whether a given heap is well-formed, i.e. whether the fields of allocated objects point to allocated objects. $\$Heap$ denotes the current value of the global heap. Allocatedness of objects is tracked by means of a special boolean field named $\$allocated$.

$[\![E]\!]_{h_1,h_2,r}$ denotes the translation of the side-effect free expression E to first-order logic, where h_1 denotes the heap, h_2 denotes the pre-state heap (used in the translation of old expressions), and r denotes the term to be substituted for the variable **result**. We will omit the second and third parameter for single-state predicates.

Pure Methods We treat pure methods as follows. For every pure method

```
pure t m(t_1 x_1, ..., t_n x_n)
  requires P; reads R; ensures Q;
{ return E; }
```

defined in a class C, a function symbol $\#C.m$ is introduced in the verification logic that takes a heap, the receiver and the method parameters as its formal parameters. To define the function symbol's meaning, three kinds of axioms are generated: an implementation axiom, a framing axiom, and a postcondition axiom.

(1) The *implementation axiom* declares that the result of the function $\#C.m$ is equal to evaluating the method body.

$$\begin{aligned}&\forall heap, o, x_1, \ldots, x_n \bullet wf(heap) \wedge heap[o, \$allocated] \wedge [\![P]\!]_{heap} \Rightarrow\\ &\quad \#C.m(heap, o, x_1, \ldots, x_n) = [\![E]\!]_{heap}\end{aligned}$$

The implementation axiom can only be used within the module where C is defined.

(2) The *framing axiom* states that the function $\#C.m$ only depends on locations in m's footprint R. More specifically, a state change does not affect the return value of m if m's precondition holds in the pre and post-state and if locations in m's footprint have equal values.

$$\begin{aligned}&\forall heap_1, heap_2, o, x_1, \ldots, x_n \bullet wf(heap_1) \wedge wf(heap_2) \wedge\\ &\quad heap_1[o, \$allocated] \wedge heap_2[o, \$allocated] \wedge [\![P]\!]_{heap_1} \wedge\ [\![P]\!]_{heap_2} \wedge\\ &\qquad (\forall q, f \bullet (q, f) \in [\![R]\!]_{heap_1} \Rightarrow heap_1[q, f] = heap_2[q, f]) \Rightarrow\\ &\qquad\quad \#C.m(heap_1, o, x_1, \ldots, x_n) = \#C.m(heap_2, o, x_1, \ldots, x_n)\end{aligned}$$

The framing axiom can only be used by modules that use the module where C is defined.

(3) The *postcondition axiom* axiomatizes the pure method's postcondition.

$$\begin{aligned}&\forall heap, o, x_1, \ldots, x_n \bullet wf(heap) \wedge heap[o, \$allocated] \wedge [\![P]\!]_{heap} \Rightarrow\\ &\quad [\![Q]\!]_{heap, heap, \#C.m(heap, o, x_1, \ldots, x_n)}\end{aligned}$$

m's postcondition axiom can only be used by modules that use the module where C is defined. For each dynamic frame, a default postcondition axiom is added stating that the dynamic frame only contains allocated objects. This axiom holds because of the well-formedness of the heap which implies that only allocated objects are reachable from allocated objects.

Our verifier prototype determines automatically which modules are being used within a certain method implementation by looking at the declared type of fields, parameters and local variables. A module is never considered to use itself.

Footprints Method footprints are enforced differently for mutators and for pure methods. For a pure method m with footprint R, we check that every location (directly or indirectly) read by the method body is an element of R. More specifically, we check at each field access and method invocation within the body that the set of objects read by the subexpression is a subset of R. For a field access $o.f$, the set of read locations equals the singleton $\{(o, f)\}$. To determine the set of locations read by a callee, we rely on the callee's reads annotation.

The footprint W of a mutator m is checked by means of an additional postcondition: for each location (o, f), either m does not affect the value of the (o, f), or o was not allocated in the method pre-state, or the location was an element of W.

$$\begin{aligned}&\forall o, f \bullet \mathbf{old}(\$Heap[o, f]) = \$Heap[o, f] \vee\\ &\quad \neg\mathbf{old}(\$Heap[o, \$allocated]) \vee\\ &\quad (o, f) \in [\![W]\!]_{\mathbf{old}(\$Heap)}\end{aligned}$$

This postcondition is used to enforce the footprint and to verify client code.

Soundness The soundness of our approach rests on two pillars: (I) the consistency of the verification logic and (II) the property that the value of a pure method is preserved by a state change, provided the footprint of the state change is disjoint from the pure method's footprint.

To satisfy the former component, we must ensure that the axioms generated based on user-defined pure methods are consistent. One way to enforce this consistency is to impose two restrictions: the module usage relation is acyclic and pure methods only call pure methods in used modules. Enforcing these restrictions guarantees termination of pure methods, which ensures (1) that implementation axioms are consistent, (2) that postconditions and reads clauses have to be proven (as there is always a path leading to the post-state), and (3) that the proof of a postcondition/framing axiom cannot rely on itself.

To show (II) we argue informally as follows. The postcondition of a mutator ensures that the method cannot modify allocated locations outside of its footprint. Similarly, by checking that every subexpression of the body of a pure method reads a subset of the method's footprint, we know that the value of a pure method depends only on locations within its footprint. Suppose the mutator m writes X, that the pure method p reads Y, and that X and Y are disjoint. Since method footprints can only contain allocated locations, m cannot modify locations within Y, since doing so would violate its writes clause. Hence, the value of p is preserved. Note that p is preserved, even if X and Y are no longer disjoint in m's post-state, since m can only write to the pre-state footprint X.

3 Invariants, Aggregates and Peers

In this section, we demonstrate how various object-oriented programming and specification patterns can be handled using our approach. More specifically, we focus on object invariants, aggregate objects and peer objects. It is important to note that supporting these patterns does *not* require any additional methodological machinery. All the examples shown in this paper have been verified automatically using our verifier prototype. Both the prototype and the examples can be downloaded from the authors' homepage [1].

Object Invariants An object invariant describes what it means for an object to be in a consistent state. For example, consider class *ArrayList* in Figure 2. An *ArrayList* object o is consistent if *o.items* points to a non-null array, and if *o.count* is a valid index in the array.

Some other approaches such as [6, 7] treat object invariants differently from regular predicates, thereby introducing additional complexity. In our approach an object invariant is just another pure, boolean method. For instance, in class *ArrayList* the method *invariant* specifies the object invariant. To assume/assert the object invariant, it suffices to assume/assert that the invariant method returns *true*.

Some readers may have noticed the peculiar, conditional form of *invariant*'s footprint. If *o.invariant*() returns true, then it is framed by *o.footprint*(); otherwise, *o.invariant*() may depend on any location (*universe* denotes the set of all locations). It suffices to frame *o.invariant*() only when it returns true, since client code that relies on the reads clause typically only "sees" valid *ArrayList* objects. Instead of using a conditional reads clause, one could frame *invariant* by *footprint*(). However, one would also have to remove *footprint*'s precondition, and modify *footprint*'s body to take into account invalid object states, thereby essentially duplicating the part of the invariant.

Aggregate Objects Many objects internally use other objects to help represent their internal state. Such objects are sometimes called aggregate objects. Typically, the consistency of an aggregate object implies the consistency of all its helper objects, and an aggregate object's footprint includes the footprint of all its helper objects. Consider the class *Stack* shown in Figure 2. A *Stack* object internally uses an *ArrayList* object to represent its internal state, and can therefore be considered an aggregate object. A stack's footprint includes its arraylist's footprint, and a *Stack* object's invariant implies the invariant of the internal *ArrayList* object.

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 a reference to its internal *ArrayList* to client code. However, in that case client code will not be able to establish disjointness between the aggregate object's footprint and the helper object's footprint. As a consequence, updating the helper object causes the client to lose all information (given by the return values of its pure methods) about the aggregate object, and as a result clients cannot falsely assume that the state of the aggregate object is preserved when one of the helper objects is modified.

The return value of pure methods can change over time. In particular, locations can be added to or removed from an object's footprint. For example, the method *Switch* (in class *Stack*) shown below exchanges the internal *ArrayList* of the receiver and the parameter *other*. Again, our approach does not impose special methodological rules to ensure this "ownership transfer" takes place safely.

```
void Switch(Stack other)
  requires other ≠ null ∧ other.invariant();
  requires invariant() ∧ footprint() ∩ other.footprint() = ∅;
  writes footprint() ∪ other.footprint();
  ensures invariant() ∧ other.invariant();
  ensures size() = old(other.size()) ∧ other.size() = old(size());
  ensures footprint() ∩ other.footprint() = ∅;
  ensures (footprint() ∪ other.footprint()).newElemsFresh();
{ ArrayList tmp = contents; contents = other.contents; other.contents = tmp; }
```

Peer Objects The examples considered so far can be verified using traditional ownership-based solutions, since the object graph has an hierarchical, tree-like structure. However, many object-oriented programming patterns, including the

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

  ArrayList()
    writes $\emptyset$;
    ensures $invariant() \wedge size() = 0$;
    ensures footprint().isFresh();
  { items := new Object[10]; }

  void add(Object o);
    requires invariant();
    writes footprint();
    ensures invariant();
    ensures $size() = \mathbf{old}(size() + 1)$;
    ensures $get(size() - 1) = o$;
    ensures $\forall i \in (0 : size() - 1) \bullet get(i) = \mathbf{old}(get(i))$;
    ensures footprint().newElemsfresh();
  { ... }

  pure Object get(int i);
    requires $invariant() \wedge 0 \leq i < size()$;
    reads footprint();
  { return items[i]; }

  pure int size();
    requires invariant();
    reads footprint();
    ensures $0 \leq \mathbf{result}$;
  { return count; }

  pure bool invariant()
    reads invariant()?footprint() : universe;
  { return $items \neq null \wedge 0 \leq count \leq items.length$; }

  pure set footprint()
    requires invariant();
    reads footprint();
  { return $\{\&count, \&items\} \cup \mathbf{elems}(items)$; }
}
```

```
class Stack {
  ArrayList contents;

  Stack()
    writes $\emptyset$;
    ensures $invariant() \wedge size() = 0$;
    ensures footprint().isFresh();
  { contents := new ArrayList(); }

  void Push(Object o)
    requires invariant();
    writes footprint();
    ensures invariant();
    ensures $size() = \mathbf{old}(size()) + 1$;
    ensures footprint().newElemsFresh();
  { contents.add(o); }

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

  pure bool invariant()
    reads invariant()?footprint() : universe;
  {
    return $contents \neq null \wedge$
      $contents.invariant() \wedge$
      $\&contents \notin contents.footprint()$;
  }

  pure set footprint()
    requires invariant();
    reads footprint();
  { return $\{\&contents\} \cup contents.footprint()$; }
}
```

Fig. 2. A class *ArrayList* and a class *Stack*. *Stack* objects internally use *ArrayList* objects.

observer and iterator pattern, do not follow this structure. For example, consider class *iterator* in Figure 3. No single iterator uniquely "owns" the list, and similarly the list does not own its iterators.

Modifying a list while iterators are iterating over it can give rise to unexpected exceptions. For example, removing elements from a list can cause an *ArrayOutOfBoundsException* in a corresponding iterator's *next* method. However, since the reads clause of an iterator's invariant includes the footprint of the corresponding list, any modification to the list immediately invalidates its corresponding iterators, making it impossible to use an iterator which is "out of sync" with its list.

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

  Iterator(ArrayList l)
    requires l ≠ null ∧ l.invariant();
    writes ∅;
    ensures invariant();
    ensures list() = l;
    ensures footprint().isFresh();
  { list := l; }

  Object next()
    requires invariant() ∧ hasNext();
    writes footprint();
    ensures invariant() ∧ list() = old(list());
    ensures footprint().newElemsFresh();
  { return list.items[index + +]; }

  pure bool hasNext()
    requires invariant();
    reads footprint() ∪ list().footprint();
  { return index < list.count; }

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

  pure bool invariant()
    reads invariant()?
      (footprint() ∪ list().footprint()) : universe;
  { return list ≠ null ∧ list.invariant()∧
    0 ≤ index ≤ list.count∧
    &list ∉ list.footprint()∧
    &index ∉ list.footprint(); }

  pure set footprint()
    requires invariant();
    reads footprint();
  { return {&list, &index}; } }
```

Fig. 3. The iterator pattern.

4 Inheritance

Adding inheritance to SJava complicates the handling of pure methods, since inheritance allows binding method calls statically and dynamically, depending on the method itself and on the calling context. More specifically, an abstract method is always dynamically bound, while a private or final method is always statically bound. Non-abstract, non-private methods can either be statically or dynamically bound: a super call to such a method is statically bound, but any other call is dynamically bound.

To model the fact that a call to a pure method m in a class C can be both statically and dynamically bound, we introduce two function symbols for every pure method in the verification logic (instead of only one): $\#C.m$ and $\#C.m_D$ (similar to [7, 8]). The former function symbol represents statically bound calls to m, and is axiomatized as described in Section 2. The latter function symbol represents dynamically bound calls is axiomatized by relating it to the former symbol using the following axiom.

$$\begin{array}{l}\forall \mathit{heap}, o, x_1, \ldots, x_n \bullet \mathit{typeof}(o) = C \Rightarrow \\ \quad \#C.m_D(\mathit{heap}, o, x_1, \ldots, x_n) = \#C.m(\mathit{heap}, o, x_1, \ldots, x_n)\end{array}$$

That is, given that the dynamic type of some object o (denoted by $\mathit{typeof}(o)$) is C, one may assume that dynamically bound calls to the object equal statically bound calls to C's method m. In addition, whenever a method $D.m$ overrides a method $A.m$, the following axiom is added: two dynamically bound calls of m yield the same result whenever the receiver's dynamic type is a subtype (denoted by $<:$) of D.

$$\begin{array}{l}\forall \mathit{heap}, o, x_1, \ldots, x_n \bullet \mathit{typeof}(o) <: D \Rightarrow \\ \quad \#A.m_D(\mathit{heap}, o, x_1, \ldots, x_n) = \#D.m_D(\mathit{heap}, o, x_1, \ldots, x_n)\end{array}$$

Calls on the receiver object in method contracts are treated differently from such calls in code. If a method call is dynamically bound, then calls on the receiver object in the method contract are treated as dynamically bound; otherwise calls in the contract on the receiver are treated as statically bound. Methods themselves are verified assuming they are called statically, i.e. calls in the contract on the receiver are bound statically. Doing so is sound, provided every subclass overrides each method. Indeed, if a method is called statically, then the caller and callee agree on the method contract. If a method is called dynamically, then the dynamic type of the receiver equals the static type, and therefore the static contract equals the dynamic one.

To ensure Liskov's substitution principle, we impose the restriction that overriding methods must inherit the contract of overridden methods as is. The only exception to this rule are postconditions. More specifically, an overriding method may extend the contract of the overridden method with additional postconditions. More flexible approaches to ensure proper subtyping exist (e.g. [9]), and combining them with our approach is part of future work.

5 Discussion

Defaults The examples shown in this paper contain on about 2 lines of specification for every line of code, where we consider the invariant and footprint methods to be part of the specification. To reduce this annotation overhead, we propose using defaults for common programming and specification patterns. More specifically, we propose generating footprint and invariant methods based on field modifiers, and adding default contracts to methods. Using these defaults reduces the number of annotation in class *Stack* of Figure 2 from 17 to 3.

The scheme is as follows. The footprint and invariant methods in a class C are generated based on C's fields. That is, the footprint method includes locations corresponding to C's fields. Moreover, fields may be marked with a **rep** modifier. The footprint of an object referenced from one of C's rep fields is also included in the footprint. Finally, C's footprint method includes the footprint of the superclass. C's invariant method states that rep fields are non-null, that the footprints of rep objects and of the superclass do not contain locations corresponding to fields of C, and that those footprints are mutually disjoint. The footprint method requires the invariant and reads itself. The invariant method reads the footprint, provided it returns true.

Each method is given a default method contract. Pure methods require the invariant and read the footprint. Constructors write the new object's footprint, ensure the invariant, and ensure that the footprint contains only fresh locations. Other mutators require and ensure the invariant, write the pre-state footprint, and ensure that the footprint is only extended with fresh locations.

Experience Table 1 lists the time taken to discharge the verification conditions generated for each program. The experiments have been carried out on a regular desktop pc with a Pentium 4 3.00 Ghz CPU and 512 Mb of RAM.

The verifier prototype is a custom build of the Spec# program verifier [10], and uses 2 theorem provers: Z3 and Simplify. The latter prover is only used if the former fails to verify a verification condition. Z3 is typically faster than Simplify, but is sometimes unable to prove the constructor's postcondition stating that elements in the new object's footprint are fresh.

	cell	**fraction**	**list, stack & iterator**	**observer**	**masterclock**
# lines	20	52	138	85	74
time taken	0.6	1.6	25.2	15.1	11.4

Table 1. Table showing the time taken (in seconds) to verify the examples. Examples not shown in this paper can be downloaded from [1].

6 Related Work

The approach presented in this paper was inspired by the work of Kassios [3, 4]. 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, higher-order 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. Moreover, we extend his solution to deal with Java-like inheritance.

In the basic Boogie methodology [6], 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.

In [11] and [12], the authors 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.

[7, 13] and [14] 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. To ensure the consistency of the verification logic (regardless of the addition of axioms generated based on pure methods), [7] and [14] enforce the termination of pure methods. The former approach does so by checking the acyclicity of the call-graph at load-time. The latter approach relies on a heuristic for finding a well-founded order. One of the major differences between their approach and ours is that we allow pure methods to depend on any location, as long as the location is an element of the method's footprint (which can be any expression of type set), and it is up to client code to track disjointness of method footprints, while they only allow pure methods to depend on objects in a limited number of ownership cones (indicated by means of rep modifiers on fields).

Leino and Müller [15] 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.

Parkinson and Bierman [16, 17] extend separation logic to the Java programming language, introducing abstract predicates to attain data abstraction. Their solution has not been implemented in an automatic program verifier, and the feasibility of automatic verification has not been shown. Furthermore, Parkinson's abstract predicates are not part of the programming language itself, while pure methods are. This might make it easier for programmers to use pure methods.

In [18], the authors propose using data groups to specify side-effects. A data group represents a set of variables (similarly to our footprint methods), and mutator methods can abstractly specify their footprint using a modifies clause in terms of these data groups (similarly to our writes clauses which use footprint methods). However, to ensure the soundness, their approach imposes two methodological restrictions: the pivot uniqueness and owner exclusion restriction. Our approach requires no such restrictions, and as a consequence it can handle programs that [18] cannot. For example, the former restriction rules out sharing of representation objects, as is the case in the iterator example shown in Figure 3.

Banerjee, Naumann and Rosenberg [19] propose using regions, state-dependent expressions similar to our method footprints, to specify the effect of mutators. They develop a Hoare-style calculus and proof its soundness. However, the logic is not implemented and, in effect, there are some challenges to do so when one would target SMT solvers, e.g. they use ghost state and "recursion" to express reachability, and they use frame subsumption, which should only be applied on demand.

Müller [8]'s thesis 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 3 were a model field, then *hasNext* would have to be visible to the class ArrayList. Our approach has no such restriction.

7 Conclusion

In summary, this paper proposed an approach to the automatic verification of Java-like programs that combines the use of pure methods to achieve data abstraction with Kassios' solution to the framing problem [3, 4]. More specifically, we solve the framing problem by extending each method contract with a method footprint, an upper bound on the set of memory locations read or written by the corresponding method. Thanks to the use of dynamic frames, pure methods that return a set of memory locations, these method footprints can be specified without breaking information hiding. The approach has been implemented in a prototype [1], which has been used to automatically verify several challenging programs, including the iterator and observer pattern. We plan to extend our approach to concurrent programs, and apply our approach in a larger case-study.

Acknowledgments

The authors thank Rustan Leino and David Naumann for helpful comments and discussions. 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).

References

1. `http://www.cs.kuleuven.be/~jans/DFJ`
2. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. In: Formal Aspects of Computing
3. Kassios, Y.: A Theory of Object Oriented Refinement. PhD thesis, University of Toronto (2006)
4. Kassios, Y.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Formal Methods. (2006)
5. Leino, K.R.M., Schulte, W.: A verifying compiler for a multi-threaded object-oriented language. In: Marktoberdorf Summer School Lecture Notes. (2006)
6. Barnett, M., DeLine, R., Fahndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology **3**(6) (2004)
7. Jacobs, B., Piessens, F.: Verification of programs with inspector methods. In: FTFJP. (2006)
8. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. PhD thesis, FernUniversität Hagen (2001)
9. Dhara, K.K., Leavens, G.T.: Forcing behavioral subtyping through specification inheritance. In: ICSE. (1996)
10. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: An overview. In: CASSIS. (2004)
11. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: ECOOP. (2004)
12. Barnett, M., Naumann, D.A.: Friends need a bit more: Maintaining invariants over shared state. In: MPC. (2004)
13. Jacobs, B., Piessens, F.: Inspector methods for state abstraction. Journal of Object Technology **6**(5) (2007)
14. Darvas, A., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: FASE. (2007)
15. Leino, K.R.M., Müller., P.: A verification methodology for model fields. In: ESOP. (2006)
16. Parkinson, M.: Local Reasoning for Java. PhD thesis, University of Cambridge (2005)
17. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: POPL. (2005)
18. Leino, K.R.M., Poetzsch-Heffter, A., Zhou, Y.: Using data groups to specify and check side effects. In: PLDI. (2002)
19. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Unpublished. (2007)