Dynamic Frames and Automated Verification

Ioannis T. Kassios

July 11, 2011

Abstract

The specification of properties that are preserved during the execution of a procedure, is an instance of the well-known *frame problem*. The frame problem has proved to be one of the thorniest obstacles in the field of Specification and Verification of programs with rich heap structures and encapsulation.

Dynamic Frames is a formalism that was developed to deal with this problem. Its emphasis is on preserving the *modularity* of the specification methodology, i.e., the ability to handle small parts of a larger program independently of each other, without compromising its *expressiveness*, i.e., without imposing a strict methodology that would exclude useful programs. Further developments focused on *automating* the verification of programs written in the Dynamic Frames style. Automation poses extra challenges, and calls for variations of the original formalism.

The original formalism as well as two variations are discussed in this tutorial. During the discussion, we compare the strengths and weaknesses of each approach, and we mention several important design goals and trade-offs, in the hope of providing a good starting point for people who are interested in conducting research in the area.

1 Introduction

Imperative programs with rich heap structures and encapsulation are mainstream, since they are featured in very popular object oriented languages such as Java and C#. We are interested in the *modular automated full functional verification* of such programs. This means that we want to specify unambiguously the desired behavior of our programs in an expressive mathematical language, and to provide a mechanism that formally proves their compliance. Furthermore, we want to do this in a scalable way, therefore our specifications must be as modular as possible, and we want our proof mechanism to be as automated as possible.

All these requirements compose a complex research domain. In particular, the *frame problem* [29] has been one of the hardest and most interesting research problems in this area. The formalism of Dynamic Frames [17, 18, 19] was invented to address this problem, initially in a theoretical setting, with

no concern for automation. Further research attempts to integrate Dynamic Frames in automated verifiers revealed various trade-offs and challenges.

In this tutorial, we explain these research challenges, and we show two different ways in which the automation of Dynamic Frames has been implemented. The tutorial can serve two purposes. One the one hand, it can be used as a case study of formal design, even by people who are not interested in Dynamic Frames per se. On the other hand, it can be used by people interested in Dynamic Frames, either as users or researchers, to understand the advantages and shortcomings of each of its variants, the state-of-the-art, and some current research problems.

2 Basics of Modular Specification

One of the most important design goals for a theory of program verification is *modularity*. The idea is that a proof of correctness can be broken down into small parts, one for each program module, which are independent of each other. This way, extensions to the program do not break already verified code. Furthermore, a change in one module invalidates only one proof.

The concept of modularity of verification is of course inspired by the same concept in programming language design. Designers of verification systems take advantage of the kind of modules provided by the targeted programming language.

In these notes, we follow a minimal style of specification for a small object based language. Our modules are *classes*. A class specification consists of pre- and postconditions attached to the methods of a class. We follow the so-called "partial correctness" semantics for methods, which means that we do not take into consideration executions that do not terminate. Subclassing is completely ignored.

Each class C has a specification S. The proof of correctness of C versus S does not depend on any other proof of correctness. When another class C' is involved in C, then we only rely on the specification $S'$of C'. This means that our proof of correctness does not break if the implementation of C' changes, provided of course that the new implementation of C' is also correct with respect to S'.

2.1 Procedural Abstraction

Fig. 1 shows an example of a correct class in our language. There is an integer field x, and a method i whose purpose is described by its postcondition: to increase the value of x (the keyword **old** may be used at the postcondition to evaluate an expression at the pre-state). The implementer is free to chose any way of satisfying that goal. Here, the implementation adds 1 to the value of x.

The modularity requirement appears in our example as follows: changing the implementation of i to, e.g., $x := x + 10$; does not invalidate the proof of correctness of any client. To satisfy this, clients must not rely on the fact that

```
class C
{
  var x : int;

  method i()
    ensures x > old(x);
  { x := x + 1; }
}
```

Figure 1: A simple class

```
class Client
{
  method m0(c : C)
    ensures c.x > 2 * old(c.x);
  { c.x := 2 * c.x; c.i(); }

  method m1(c : C)
    ensures c.x = 2 * old(c.x) + 1;
  { c.x := 2 * c.x; c.i(); }
}
```

Figure 2: A client for C

i increases x by 1: this implementation detail might change. The clients must rely only on the specification $x > \mathbf{old}(x)$. For example, in Fig. 2, the method $m0$ is correct, but the method $m1$ is not correct.

We often refer to this policy as *procedural abstraction*: roughly the client is told "what" is being computed, but "how" this is computed remains hidden.

2.2 Data Abstraction

Most of the time, the implementer wants to hide from the client not only the actual code, but also the actual *data representation* of an implementation. For example, we might want a class that represents *sequences* of integers. There are various data structures that can be used for that purpose, e.g., arrays, linked lists, doubly linked lists etc. The actual representation is of no interest to the client, who is only interested in using the abstract mathematical concept of a sequence. The implementer should be free to change the data representation without breaking the verification of the client. This idea is called *data abstraction* [15].

The client must not be able to refer to the fields of a class at all: if the implementation changes, then a field may disappear, or, worse yet, represent something different. But then, we have a problem: how can we write specifications that the client can even understand? For example, in Fig. 1, if x is meant to be hidden, the specification of i is not legal: the client does not know x.

The idea advocated in [15] is to invent a new vocabulary for the client, that of *abstract variables*. The specifications are written using visible abstract variables, which the client can understand. These variables are used in the reasoning of the client. On the other hand, the implementer is using *concrete variables* in the implementation.

In object orientation, the vocabulary that the client uses to refer to the state of an object is the *observer* methods of a class, i.e., methods which return a value that depends on the actual state of the object. It is natural to use the observer methods as abstract vocabulary. The implementation of the observer methods connects the two vocabularies.

To simplify this idea, we introduce the construct of *pure functions*. These are members of a class whose invocation computes a result without changing the state. Their body is an expression of the programming language. We allow invocations of pure functions to appear in specifications of methods. In Fig. 3, we implement a class that represents *sequences* of integers using a hidden linked list and specify its methods using the provided pure functions *get* and *len*.

This style of specification is attractive because of its simplicity. Notice however that the possibility of non-termination of the evaluation of a pure function may lead to inconsistencies. For example, the following definition introduces an inconsistent axiom:

function $f()$ **returns bool** { $\neg f()$; }

In the example of Fig. 3, the function *len* has inconsistent definition when the list is cyclic.

Our verification technique must show that the evaluation of any pure function terminates, when done in a state that satisfies its precondition. This means that it must provide somehow an upper bound to the evaluation time [14]. We come back to the problem later, but we leave the specification as it is now.

3 The Frame Problem and Dynamic Frames

3.1 The Frame Problem

As proud as we are of our classes in Fig. 1 and 3, the sad truth is that they are of no use to any real client. The logical problem does not have to do with what the methods of our class do, but with what they *don't do*.

For example, consider the client of the *List* class in Fig. 4. After constructing two *List* objects, A, B (in that order), the client expects A to represent the empty list. This expectation is expressed by the **assert** statement. Such a statement, called an *assertion*, must be proved to be correct in any execution of the program.

The specification of *List* is too weak for the client to verify the assertion. The problem happens after executing the statement $B :=$ **new** *List*;. The specification of the constructor of *List* promises that the new object B is going

```
class Node
{
  public var v : int;
  public var n : Node;
}

class List
{
  private var c : Node;

  List()
    ensures len() = 0;
  { c := null; }

  public method insert(x : int)
    ensures len() = old(len()) + 1 ∧ get(0) = x;
    ensures ∀i : 1..len() − 1 · get(i) = old(get(i − 1));
  {
    var p : Node;
    p := new Node; p.v := x; p.n := c;
    c := p;
  }

  public function get(i : int) returns int
    requires 0 ≤ i < len();
  { get_aux(i, c); }

  private function get_aux(i : int, p : Node) returns int
    requires 0 ≤ i < len_aux(i, p);
  { i = 0 ? p.v : get_aux(i − 1 , p.n); }

  public function len() returns int
  { len_aux(c); }

  private function len_aux(p : Node) returns int
  { p = null ? 0 : 1 + len_aux(p.n); }
}
```

Figure 3: The *List* class

```
var A, B : List;
A := new List; B := new List;
assert A.len() = 0;   //fails
```

Figure 4: An incorrect client of *List*.

to represent the empty list, but makes no promise about what else may happen to the state during the construction of B. In particular, we have no guarantee that $A.c$ will not change.

The problem can be stated as follows: *when formally describing a change in a system, how do we specify what parts of the state of the system are not affected by that change?* This is called the *frame problem* [29].

The instance of the frame problem that we are dealing with is quite hard. The specifier cannot express logically that "no client variable is affected". In our modular setting, the variables of the client are unknown. Neither can the specifier express the fact that only the field c of the new object is modified. Remember that c is invisible to the client, who can observe the state of the object only through the pure functions *get* and *len*.

One could think that we can add "modifies" clauses to our methods, that mention pure functions. For example the method *insert* modifies the pure functions *len* and *get* of the target object, leaving everything else (and therefore the pure functions of other objects) unchanged. Such reasoning would in general be unsound, because of a situation called *abstract aliasing* [28]: A and B may *share* some nodes in their secret linked list implementation, which means that an operation on the nodes of B may affect the internal state of A, unbeknownst to the client. Abstract aliasing cannot happen in our implementation of Fig. 3, but the specification does not reflect that. In fact, our specification language so far has *no way of expressing* the possibility or absence of abstract aliasing.

3.2 Dynamic Frames

The idea behind Dynamic Frames, is the introduction in the abstract vocabulary of the *footprints* of methods and functions. The footprint of a method is the set of all fields that the method is permitted to modify. The footprint of a pure function is the set of all fields that the function is permitted to read.

If footprints can be expressed in the abstract vocabulary, we can express absence of interference by proving that the footprint of a method is disjoint from the footprint of a pure function. In the example of Fig. 4, we need to prove that the footprint of the constructor of B is disjoint from the footprint of $A.len()$, at the point where the constructor of B is called.

The footprint of a method is written in a "modifies" clause in the specification of that method. In particular, **modifies** F; means that the method is allowed to modify only the fields that are included in F *evaluated in the pre-state*, or fields of newly allocated objects.

The footprint of a function is written in a "reads" clause in the specification of that function. In particular, **reads** F; means that the function depends only on fields included in F.

To express footprints abstractly, we introduce pure functions whose evaluation yields a set of fields. Such a pure function is called a *dynamic frame*. We denote by **reg** (*region*) the return type of dynamic frames.

We can now annotate with footprints the *List* class. We need one public dynamic frame *rep* (*representation region*) that contains all the nodes of the

list. Fig. 5 shows how the new specification looks. The implementations of the methods are omitted.

The constructor has no "modifies" clause, which defaults to an empty footprint. This is correct: all the constructor modifies is the field c of the newly allocated object, which need not be included in the footprint. The method *insert* has *rep*() as its footprint.

Having dynamic frames, we can write specifications that were impossible in our previous language. For example, we can add the requirement that there are no cycles in our list. This is an *invariant* of the list object. We express it as a pure function *inv*() that returns a boolean. Fig. 6 shows the definition of the invariant. The requirement of disjointness of frames ensures acyclicity. We add *inv*() as a precondition and postcondition to all our methods and as a precondition to all other pure functions. We add *inv*() only as postcondition in the constructor.

We can also write and specify a method *prepend* which takes another list argument and prepends it to the current list. The method has the precondition that the representation regions of both list objects are disjoint, which is now expressible. Fig. 7 shows the specification and the implementation of the *prepend* method.

3.3 Swinging Pivots and Self-Framing

So far our notation gives a way for the specifier to describe in the abstract level an overapproximation of all fields that a method changes and all fields that a pure function depends on. We also have managed to express non-interference properties necessary to introduce a new method and an invariant in our class. The implementer has the way to define this abstract vocabulary nicely.

It seems that our work is done. However, this is not so. Fig. 8 shows an example of the problem that remains to be solved.

The client starts with two lists A, B that have no abstract aliasing with each other. For the sake of the example, we know that $B.len() = 1$ in the prestate.

Our first statement inserts 10 to A. Since the footprint of $A.insert(10)$ is initially disjoint from that of $B.len()$, we know that $B.len()$ preserves its value through this operation, and the first assertion succeeds. However, after we perform the insert operation for the second time, the expression $B.len() = 1$ cannot be proved to be true anymore. What happened?

The problem is that in the state between the two calls to $A.insert(10)$ we cannot prove the non-interference property $A.rep() \cap B.rep() = \emptyset$. The property cannot be proved, because the first invocation of $A.insert(10)$ does not guarantee anything about how the values $A.rep()$ or $B.rep()$ change.

To solve this problem, we must have a methodology that guarantees that *when a method is invoked, all the disjointness properties of the dynamic frames it does not know about are preserved.* In our example, the specification of *insert* should be such that if the representation region of the current object is disjoint from an unknown dynamic frame, then it remains disjoint after the invocation of *insert*. This is not trivial, because we cannot directly refer to dynamic frames

```
class List
{
  private var c : Node;

  List()
    ensures len() = 0;
  { c := null; }

  public method insert(x : int)
    modifies rep();
    ensures len() = old(len()) + 1 ∧ get(0) = x;
    ensures ∀i : 1..len() − 1 · get(i) = old(get(i − 1));
  { ... }

  public function rep() returns reg
  { {this.c} ∪ rep_aux(c); }

  private function rep_aux(p : Node) returns reg
  { p = null ? ∅ : {p.v, p.n} ∪ rep_aux(p.n); }

  public function get(i : int) returns int
    requires 0 ≤ i < len();
    reads rep();
  { get_aux(i, c); }

  private function get_aux(i : int, p : Node) returns int
    requires 0 ≤ i < len_aux(i, p);
    reads rep_aux(p);
  { i = 0 ? p.v : get_aux(i − 1 , p.n); }

  public function len() returns int
    reads rep();
  { len_aux(c); }

  private function len_aux(p : Node) returns int
    reads rep_aux(p);
  { p = null ? 0 : 1 + len_aux(p.n); }
}
```

Figure 5: The *List* class with dynamic frames

```
public function inv() returns bool
  reads rep();
{ inv_aux(c); }

private function inv_aux(p : Node) returns bool
  reads rep_aux(p);
{ p = null ∨ ({p.v, p.n} ∩ rep_aux(p.n) = ∅ ∧ inv_aux(p.n)); }
```

Figure 6: The *inv* function

```
public method prepend(p : List)
  requires p ≠ null ∧ inv() ∧ p.inv();
  requires rep() ∩ p.rep() = ∅;
  modifies rep() ∪ p.rep();
  ensures len() = old(len() + p.len());
  ensures ∀i : 0..old(p.len()) − 1 · get(i) = old(p.get(i));
  ensures ∀i : old(p.len())..len() − 1 · get(i) = get(i − old(p.len()));
  ensures inv();
{
  var q : Node;
  if(p.c ≠ null)
  {
    q := p.c;
    while(q.n ≠ null) q := q.n;
    q.n := c;
    c := p.c;
  }
}
```

Figure 7: The *prepend* method

```
method cl(A : List, B : List)
  requires A.rep() ∩ B.rep = ∅;
  requires B.len() = 1;
{
  A.insert(10);
  assert B.len() = 1;   //succeeds
  A.insert(10);
  assert B.len() = 1;   //fails!
}
```

Figure 8: Failing to Preserve Disjointness

```
Set()
  ...
  ensures fresh(rep());
{ ... }

public method insert(x : int)
  ...
  ensures fresh(rep() − old(rep()));
{ ... }

public method prepend(p : List)
  ...
  ensures fresh(rep() ∪ p.rep() − old(rep() ∪ p.rep()));
{ ... }
```

Figure 9: Swinging Pivots Specifications for the *List* class

we do not know about. It turns out that we need two conventions: the *swinging pivots restriction* and *self-framing*.

3.3.1 Swinging Pivots

The *swinging pivots restriction* is named after a requirement of the methodology of [28], which has been relaxed significantly in the Dynamic Frames theory. The restriction, as it is now, can be expressed as follows. Let S be the *set of dynamic frames that are involved in the footprint of a method*. The value of any dynamic frame in S may be increased only by locations that are initially in some other dynamic frame in S or by newly allocated locations.

For example, in *insert*(x), we only have *rep*() in the footprint. The value of *rep*() may be increased only by newly allocated locations. In *prepend*(p), we have two dynamic frames in the footprint, *rep*() and p.*rep*(). The dynamic frame *rep*() may be increased only by locations that were previously in p.*rep*(), or newly allocated locations. Similarly, p.*rep*() may be increased only by locations that were previously in *rep*() or newly allocated locations.

We introduce a keyword **fresh**, to be used in two-state expressions, such as postconditions. It takes as argument a region-valued expression E and asserts that anything new in E is freshly allocated. Using **fresh**, the swinging pivots requirement is expressed as a postcondition to our methods as shown in Fig. 9

3.3.2 Self-framing

The swinging pivots restriction is not enough. It guarantees that the dynamic frames we know about do not step on unknown territory. But it does not guarantee that the dynamic frames we do not know about do not behave badly.

For example, in Fig. 8, after the first *insert*, we know that A.*rep*() does not "step on" B.*rep*(), because it is only increased with previously unallocated

```
public function rep() returns reg
  reads rep();
{ rep_aux(c) }

private function rep_aux(p : Node) returns reg
  reads rep_aux(p);
{ ... }
```

Figure 10: Self-framing *rep*()

locations. But we do not know how $B.rep()$ has changed.

Dynamic frames that are unknown by a method m and disjoint from its footprint, such as $B.rep()$ in this example, should not change when m is invoked. In fact, if no field within a dynamic frame is touched, then the dynamic frame itself should not change. Dynamic frames *must frame themselves*, i.e., the footprint of a dynamic frame must be the dynamic frame itself.

To make *rep*() self-framing we just give it footprint *rep*(), as in Fig. 10.

With self-framing and swinging pivots, the client in Fig. 8 should verify. By swinging pivots, the first *insert* does not allow $A.rep()$ to "step on" $B.rep()$. At the same time, $B.rep()$ keeps its value, thanks to self-framing. This means that the non-interference property is preserved before the second *insert*. The non-interference property guarantees that $B.len() = 1$ in the end.

4 Automated Verification

Our new focus is to verify that our implementations comply to their specification in an automated way, i.e., using a push-button tool (*an automated verifier*) for that purpose. We do not want the programmer to interact with the verifier at all: all the information that the verifier needs to do its job should already be in our specifications.

Currently, there are two popular techniques for automated program verification: the *Verification Condition Generation* (VCG) [12, 23] and the *Symbolic Execution* (SE) [20]. A VCG-verifier transforms the program together with its specifications into a big logical formula, using some program calculus (usually a weakest precondition calculus [11]), and then passes this formula on to an *automated theorem prover*, the program that performs the actual proving. VCG-verifiers include Boogie [23] and Why [12]. A SE-verifier executes *all possible program paths* from the beginning to the end using *symbolic* instead of actual values, gathers logical information information on the way, and calls the theorem prover only on demand passing it each time the logical information it collects for each state. SE-based tools are the KeY system [4], Smallfoot [5], VeriCool [38], and Verifast [16] among others.

Regardless of the technique used to generate the formulas, the process depends crucially on the theorem prover used at the back end. It seems that the soft spot for automated theorem proving at this moment in time is the so-called

SMT-solvers (*Satisfiability Modulo Theories*), such as Z3 [30] and Simplify [9], which support first order logic together with custom *theories*, which come with their own decision procedures. State-of-the-art SMT solvers have native support for linear arithmetic, can be extended by theories on demand, are reasonably fast, effective and require no interaction.

There are several issues that affect the performance of SMT solvers. These must be taken into account when designing the automation of a specification theory:

- The expressiveness of the language is limited to *first order logic* (together with some extensions, such as linear arithmetic).
- Proof obligations with *quantifiers*, although they are handled by SMT-solvers, must be avoided as much as possible, for performance reasons.
- A major issue for the axioms fed to an SMT-solver is the occurrence of *matching loops*. A typical situation that explains the problem is when we have recursive axioms of the form:

  $$\forall x \cdot \ f(x) = g(f(h(x)))$$

  Suppose that the prover is trying to work on (e.g., prove) a formula that contains $f(3)$. One possibility is to instantiate the above quantification for $x = 3$ and get $g(f(h(3))$. The new term contains the subterm $f(h(3))$, which means that the axiom can be instantiated again. This can go on forever, in which case the prover fails to terminate.

  Recursive definitions such as the above are very frequent in programming, therefore matching loops show up very often in program verification.

5 Automating Dynamic Frames

There are several issues with using SMT-based tools to prove programs specified in the Dynamic Frames style. For example, it is not clear how to prove properties involving footprints, e.g., footprint compliance and self-framing, without exact *non-recursive* definition of dynamic frames. Such a definition however would in general require *reachability* predicates, because dynamic frames are non-trivial underapproximations of the set of locations that are reachable by an object (this is also the case in our *List* example). Reachability predicates are not expressible in first order logic, and therefore cannot be used with an SMT-based tool. In this section, we present two techniques for overcoming such problems.

The first technique is to model dynamic frames, and potentially other members of the abstract vocabulary, with *ghost fields*, i.e., "imaginary" fields which do not appear in the compiled version of the program. Variations of this technique are Regional Logic [1], and the program verifier Dafny [22], which we use for demonstration.

The second technique is the use of *explicit folding and unfolding* of the recursive definitions. This technique is being primarily used in Separation Logic verification tools, and in an important variation of the Dynamic Frames theory, that of *Implicit Dynamic Frames* [38]. We use Chalice [24], an Implicit Dynamic Frames tool, for demonstration.

5.1 Ghost Fields for Dynamic Frames

The so-called *ghost fields* do not influence the execution, and thus do not appear in the compiled version of a program, but they may be assigned to and they may be used in specifications. A value that is stored in a ghost field does not need a specific definition. The recursive properties that the ghost field must satisfy are part of the invariant.

In our example, *rep* can be introduced as a ghost field. The length of the list can also be introduced as a ghost field. Such an approach solves the problem of well-definedness of the *len*() pure function. The function *get*() can stay as it is, as its evaluation can be proved to be bound by *len*, and therefore its definition is well-founded.

The language and program verifier Dafny [22] depends heavily on this use of ghost variables. In Fig. 11, we see part of the list example written in Dafny.

In Dafny, the granularity of framing is coarser than what we have been describing so far: footprints are sets of objects and not fields. Permission to read from / write to a field is granted if the corresponding object is in the footprint.

Specification Style. As discussed above, *len* and *rep* are now ghost fields, not only in the *List* class, but also in the *Node* class. We see that the invariant supports the recursive relations for these fields. The invariant is assumed as a precondition and asserted as a postcondition in method *insert*. Notice that this means that the programmer must add ghost assignments within the code of *insert*.

Definedness of Pure Functions. The keyword **decreases** indicates a termination measure for a pure function. We see here that *len* is a termination measure that guarantees that *inv*() is well-defined. Similarly, *get*(i) has a **decreases** clause of i (not shown in the code). All pure functions used in specifications must have such a **decreases** clause, which guarantees their definedness.

Footprint Compliance. One important feature of the Dafny approach is the handling of footprint compliance, i.e., the proof that all methods write at most to objects in their footprint, and that all functions read at most from objects in their footprint.

A Dafny program translates into a Boogie program with only one state variable H that contains the whole heap. The heap maps pairs of objects and fields to values. There is a special boolean field *alloc* that separates allocated

```
class Node
{
  var v : int;
  var n : Node;
  ghost var len : int;
  ghost var rep : set< object >;

  function inv() : bool
    reads this, rep;
    decreases len;
  {       this ∈ rep  ∧  null ∉ rep  ∧  len > 0
    ∧   (n = null   ⇒   rep = {this}  ∧  len = 1)
    ∧   (n ≠ null
           ⇒   n ∈ rep ∧  n.rep ⊆ rep  ∧  this ∉ n.rep  ∧  len = n.len + 1  ∧  n.inv())
  }
}

class List
{
  var c : Node;
  ghost var len : int;
  ghost var rep : set< object >;

  function inv() : bool   ...//similar to the Node class

  method Init()
    modifies this;
    ensures len = 0  ∧  fresh(rep − {this})  ∧  inv();
  { c := null;  len := 0; rep := {this};  }

  method insert(x : int)
    requires inv();
    modifies rep;
    ensures inv()  ∧  len = old(len) + 1  ∧  get(0) = x;
    ensures ∀i : int · 1 ≤ i < len  ⇒  get(i) = old(get(i − 1));
    ensures fresh(rep − old(rep));
  {
    var p : Node;
    p := new Node;  p.v := x;  p.n := c;  c := p;
    len := len + 1;  c.len := len;  rep := rep ∪ {p};
    if(c.n = null) { c.rep := {c};  } else { c.rep := {c} ∪ c.n.rep;  }
  }
  ...
}
```

Figure 11: The *List* class in Dafny

from unallocated objects. In this encoding, the clause **modifies** F; can be expressed as the following extra Boogie postcondition:

$$\forall o, f \cdot \ \mathsf{H}[o, f] = \mathbf{old}(\mathsf{H})[o, f] \ \vee \ (o, f) : \mathbf{old}(F) \ \vee \ \neg \mathbf{old}(\mathsf{H})[o, \mathit{alloc}]$$

Similarly, the clause **reads** F; in the specification of a pure function[1] $g()$ can be expressed as the following proof obligation:

$$\forall \mathsf{H}' : \mathit{Heap} \cdot \ (\forall o : F, \ f \cdot \ \mathsf{H}[o, f] = \mathsf{H}'[o, f]) \Rightarrow g()_{\mathsf{H}} = g()_{\mathsf{H}'}$$

where E_H is expression E evaluated in heap H.

However, such an encoding is bad, because it introduces proof obligations with a lot of universal quantifiers. Dafny's approach is different:

- In any method, the encoding of every assignment $o.f := E$ induces a Boogie-assertion

  assert $o : \mathbf{old}(F)$;

  where F is the footprint of the method.

- In any function, each occurrence of a field $o.f$ induces a Boogie-assertion

  assert $o : F$;

  where F is the footprint of the function.

This makes the check of footprint compliance fast, avoiding the big cost of proving quantifier-laden formulas, but, at the same time, enforces the programmer to be careful when writing function bodies. As an example, let us look at the definition of *inv*() in Fig. 11. The declared footprint is $\{\mathbf{this}\} \cup \mathit{rep}$. This might seem to be superfluous, since **this** should be included in *rep*. However, there is no such information in the precondition of the invariant. In order to be able to refer to **this** and its fields we must put it explicitly in the footprint.

Self-framing. When a ghost field is used to store the value of a dynamic frame, there is no direct definition of the dynamic frame. Self-framing is ensured by the fact that there is no explicit assignment to any unknown dynamic frame.

To demonstrate this, we show in Fig. 12 the encoding of the method *prepend* in Dafny, and in Fig. 13 a client that depends heavily on frame disjointness.

[1] For simplicity, without arguments.

```
method prepend(p : List)
  requires p ≠ null ∧ rep ∩ p.rep = ∅ ∧ inv() ∧ p.inv();
  modifies rep ∪ p.rep;
  ensures inv() ∧ len = old(len + p.len);
  ensures ∀i : int · 0 ≤ i < old(p.len) ⇒ get(i) = old(p.get(i));
  ensures ∀i : int · old(p.len) ≤ i < len ⇒ get(i) = old(get(i − p.len));
  ensures fresh(rep ∪ p.rep − old(rep ∪ p.rep));
{
  if(p.c ≠ null)
  {
    call prepend_aux(p.c); c := p.c;
    len := len + p.len; rep := {this} ∪ p.c.rep;
    assert ∀i : int · 0 ≤ i < old(p.len) ⇒ old(p.get(i)) = get_aux(i, c);
    //this assertion is used as a lemma to help Dafny
  }
}

method prepend_aux(q : Node)
  requires q ≠ null ∧ rep ∩ q.rep = ∅ ∧ inv() ∧ q.inv();
  modifies q.rep;
  decreases q.len;
  ensures q.inv() ∧ q.len = old(len + q.len);
  ensures ∀i : int · 0 ≤ i < old(q.len) ⇒ get_aux(i, q) = old(get_aux(i, q));
  ensures ∀i : int · old(q.len) ≤ i < q.len ⇒ get_aux(i, q) = old(get(i − q.len));
  ensures fresh(q.rep − old(q.rep ∪ rep));
  ensures old(q.rep) ∪ rep − {this} = q.rep ∧ this ∉ q.rep;
{
  q.rep := q.rep ∪ rep − {this}; q.len := q.len + len;
  if(q.n ≠ null){ call prepend_aux(q.n); } else { q.n := c; }
}
```

Figure 12: The method *prepend* in Dafny

```
method client()
{
  var A, B, C : List;
  A := new List; call A.Init();
  B := new List; call B.Init();
  C := new List; call C.Init();
  call A.insert(1); call B.insert(2); call A.insert(3); call B.insert(4);
  call C.insert(10); call A.prepend(B); call C.insert(20);
  assert A.len = 4 ∧ A.get(0) = 4 ∧ A.get(1) = 2;
  assert A.get(2) = 3 ∧ A.get(3) = 1;
  assert C.len = 2 ∧ C.get(0) = 20 ∧ C.get(1) = 10;
  assert A.rep ∩ C.rep = ∅;
}
```

Figure 13: A client of *List* in Dafny

Dealing with Recursion. As we see in the Dafny code, recursive definitions still exist: in particular, the pure function *inv*() is recursive. This may cause matching loops, which, as we commented above, are dangerous for the underlying SMT solver.

To deal with this problem, Dafny employs the use of so-called *limited functions*. This means that the encoding of recursive definitions into Boogie employs a low-level SMT mechanism to ensure that a recursive function definition may be used as an axiom *at most once*. The mechanism of limited functions avoids the problem, at the cost of weakening the prover. Limited functions may be switched off on demand, when a proof requires more unrollings, but this happens rarely.

Discussion. The use of ghost variables, as is supported by Dafny, is a very flexible solution for the automation of the verification of Dynamic Frames. The manual updates to specification-only variables provide extremely valuable guidance to the prover. On the other hand, it is an annotation overhead which may by itself be a source of bugs. The methods *prepend* and *prepend_aux* of Fig. 12 are cases where this overhead is very heavy.

The flexibility of the Dafny language has another price: the specifications of methods and functions become quite verbose. This is a problem with Dynamic Frame specifications in general, that Dafny does not address.

The Dafny program described in this section has been verified by the Dafny program verifier 2.0.0.0, on top of Boogie version 2.0.0.0 and the Z3 SMT solver, version 2.15.

5.2 Recursion with Folding

The next methodology that we explore is the use of *explicit folding and unfolding* of recursive definitions. The idea is as follows: suppose that we have a recursive definition $x = f(x)$. We do not allow the prover to use this definition as a rewrite rule from left to right, unless we specifically *unfold* it. Dually, we do not allow the definition as a rewrite rule from right to left, unless we specifically *fold* it.

Suppose that x, as defined above, is of boolean type. If we know x, we cannot get our hands to $f(x)$, unless we specifically unfold x. Then, we give up knowledge of x, and gain knowledge of $f(x)$. In the opposite direction, folding $f(x)$ gives up knowledge of $f(x)$ and produces knowledge of x.

Metatheoretically, we can show that this makes x behave as the *least fixpoint* of its definition: x is true if and only if it can be *proved true* by its definition. Like the invariants of Dafny, explicit folding and unfolding enables the programmer to perform inductive proofs in first order logic.

In Implicit Dynamic Frames [38], perhaps the most important variant of Dynamic Frames, the "accessibility" predicate $acc(o.f)$ is introduced. Knowledge of $acc(o.f)$ means knowledge that the field $o.f$ *is in the footprint of the method or function where it appears*. The footprint is now defined not in a **modifies** clause, but in the precondition, e.g.,

requires $acc(\mathbf{this}.x) \ \wedge \ acc(\mathbf{this}.y)$;

means that the method/function has permission to write/read the fields x and y of the current object.

A (possibly recursive) definition that contains accessibility predicates can happen in what is called an *abstract predicate* (after the related notion introduced in [32] for Separation Logic), or simply a *predicate*. The following is a typical predicate that gives access to all the nodes in a linearly linked list:

predicate *inv*
$\{ \ acc(\mathbf{this}.v) \ \wedge \ acc(\mathbf{this}.n) \ \wedge \ (\mathbf{this}.n \neq \mathbf{null} \Rightarrow \mathbf{this}.n.inv) \ \}$

It is such predicates that can be folded and unfolded.

For example, suppose that we have a linearly linked list starting from a node c, and we have *permission* (knowledge of) $c.inv$. Suppose that we want to write to $c.n.v$. We must do the following:

- Unfold $c.inv$. This loses permission to $c.inv$ and gains permission to its body. Provided that $c.n \neq \mathbf{null}$, the permission that we now have is

$$acc(c.v) \ \wedge \ acc(c.n) \ \wedge \ c.n.inv$$

- Unfold $c.n.inv$. Now we have the permission

$$acc(c.v) \ \wedge \ acc(c.n) \ \wedge \ acc(c.n.v) \ \wedge \ (c.n.n \neq \mathbf{null} \Rightarrow c.n.inv)$$

- This permission includes $acc(c.n.v)$, which is what we want. Now we can write to $acc(c.n.v)$.

The above description indicates that accessibility predicates and abstract predicates can be understood as *resources* in the sense of linear logic. The Implicit Dynamic Frames theory proceeds one step further: there is only *one* resource $acc(o.f)$ for any field $o.f$. This means that

$$acc(o.f) \ \wedge \ acc(o.f) \ \ \Leftrightarrow \ \ \mathbf{false}$$

Furthermore, suppose that we know $A \wedge B$, where A and B are arbitrary predicates. Since there is only one resource $acc(o.f)$, that resource can be folded in either A or B or none of the two, but it cannot be folded in both of them. This means that the part of the heap that A talks about is *disjoint* from that of B. Conjunction in Implicit Dynamic Frames is *separating*, as in the case of the $*$ operator of Separation Logic. This entails two things:

- The formalism *gives up* the normal "non-separating" conjunction.
- Disjointness of frames becomes *very concise*, as in the case of Separation Logic.

Chalice [24] is a language and a verifier based on Implicit Dynamic Frames. In Fig. 14, we see the *List* example written in Chalice. For simplicity, we only see the specifications that have to do with permissions. The example has been verified by Syxc [37], a Symbolic Execution tool, using the Z3 v.1.15 solver.

Specification Style. Fig. 14 is a typical example of the specification style of Implicit Dynamic Frames.

The recursive predicate *inv* gives access to the whole linked list, but must be explicitly unfolded to do so.

The specification of *insert* requires *inv* to be ensured by its environment. Since it involves a change of the field c, that predicate must be unfolded. A new node is created and assigned to c. Before folding *inv*, we must fold the invariant $c.inv$ of the new head-node.

The specification of *prepend*(p) is more complex. The specification requires both the current object and object p to be valid. The separating conjunction ensures that their invariants share no locations. The postcondition returns *inv*, but not $p.inv$. The reason is that the client is not meant to use the p object anymore. The same is true in the corresponding Dafny implementation.

Within *prepend*, access to $p.c$ is required, so $p.inv$ is unfolded. If $p.c$ is **null**, there is nothing to do, but $p.inv$ is not folded back: the method is not required to do that.

If $p.c$ is not **null**, then we have to call *prepend_aux* on $p.c$, like in the corresponding Dafny implementation. We must give it access to $p.c.inv$, and to $acc(c)$, since this method modifies c. In fact, we give it access to the unfolded version of *inv*, because the folding of $q.inv$ in the end of the method requires this to succeed.

A striking difference from Dafny and other traditional specification theories is the fact that pre/postconditions have *side-effects*, and in particular permission exchange between the caller and the callee.

Definedness of Predicates and Pure Functions. Separating conjunction is usually the argument behind the definedness of a predicate. For example, the *inv* definitions of both *Node* and *List* are in the form:

$$\text{some access permissions } \wedge \text{ recursive occurrence}$$

Given the metatheoretical invariant that there are only finitely many access permissions at any point in the execution of a program, and that the access permissions that appear to the left of $\wedge$ may not appear to its right, this definition provides a measure of termination.

```
class Node
{
  var v : int;
  var n : Node;

  predicate inv { acc(v) ∧ acc(n) ∧ (n ≠ null ⇒ n.inv) }
}

class List
{
  var c : Node;

  predicate inv { acc(c) ∧ (c ≠ null ⇒ c.inv) }

  method insert(x : int)
    requires inv;
    ensures inv;
  {
    var p : Node;
    unfold inv;
    p := new Node; p.v := x; p.n := c; c := p;
    fold c.inv; fold inv;
  }

  method prepend(p : List)
    requires inv ∧ p ≠ null ∧ p.inv;
    ensures inv;
  {
    unfold p.inv;
    if(p.c ≠ null)
    {
      unfold inv;
      call prepend_aux(p.c); c := p.c;
      fold inv;
    }
  }

  method prepend_aux(q : Node)
    requires q ≠ null ∧ q.inv ∧ acc(c) ∧ (c ≠ null ⇒ c.inv);
    ensures q.inv ∧ acc(c);
  {
    unfold q.inv;
    if(q.n ≠ null) { call prepend_aux(q.n); } else { q.n := c; }
    fold q.inv;
  }
}
```

Figure 14: The *List* example written in Chalice

We have not shown Implicit Dynamic Frame pure functions so far. Their well-definedness has the same issues as in Dafny. However, Implicit Dynamic Frames provide yet another measure of termination.

Consider the following definition of a pure function that calculates the length of the list of nodes that starts from the current node:

```
function len()
  requires inv;
{ unfolding inv in n = null ? 1 :  1 + n.len() }
```

This definition requires *inv* to be true when *len*() is evaluated. To perform the evaluation, we perform an in-place unfolding of *inv*, using the keyword **unfolding**. This permits us to check field n, and evaluate its *len* function recursively. When the evaluation ends, then *inv* is folded back.

Notice that, since *inv* is well-defined, it cannot be unfolded for ever. Therefore, a recursive call that happens withing an **unfolding** block is always safe.

Footprint Compliance. We have already explained how footprint compliance works in Chalice. The use of access predicates avoids quantified formulas. However, it requires the programmer to explain to the prover, in the sense of folding/unfolding how their program has permission to write to a field.

Self-framing. The issue of self-framing in Implicit Dynamic Frames manifests itself as follows. Suppose that we have permission to $A \wedge B$, and that we lose (e.g., through (un)folding) permission to A. We should still have permission to B. For this to be possible, and therefore for the whole methodology to be sound, A and B must be *self-framing*, i.e., any permission that is needed to evaluate them must be contained in them. Example of a self-framing expression is $acc(o.x) \ \wedge \ o.x = 10$. On the other hand, the expression $o.x = 10$ is *not* self-framing.

In a program, all specifications and all predicate definitions must be self-framing. This means that $o.x = 10$ is not allowed by itself as a precondition or a postcondition. Similarly, *o.len*() is not allowed (where *len* is the function defined above). The correct self-framing version for the latter expression is $o.inv \ \wedge \ o.len()$: if we have the precondition of *o.len*(), then we have all the permissions needed to evaluate it.

Implicit Dynamic Frames employ a purely syntactic way to check self-framing. In fact, the algorithm is so simple that it may reject obviously correct self-framing expressions. For example, $acc(o.x) \ \wedge \ o.x = 10$ is accepted, because the algorithm checks the expression from left to right. On the other hand, the theoretically equivalent $o.x = 10 \ \wedge \ acc(o.x)$ is rejected, because the algorithm has not "seen" $acc(x)$ before reaching the expression $o.x = 10$. This behavior may seem counter-intuitive, but on the other hand it is simple, sound, and efficient.

Dealing with Recursion. As we saw, recursion in predicates is dealt with using explicit folds and unfolds. On the other hand, recursion in functions may

also produce matching loops and must be somehow tamed. There is no general solution here: each tool takes its own measures.

Discussion. Implicit Dynamic Frames brings Dynamic Frames closer to Separation Logic. They notation in this theory is significantly more concise than the ghost field solution, thanks to the use of separating conjunction. Furthermore, as we discussed above, many issues, such as footprint compliance and self-framing checks, are performed very easily.

On the other hand, Implicit Dynamic Frames throws away non-separating conjunction, reverting to a form of linear logic. Such a non-standard logic, in which specifications have side-effects to the ghost state, may be not so intuitive for the specifier. It is also not known how much this loses in terms of expressiveness.

The explicit use of folds/unfolds is very helpful for the prover, and, occasionally reveals interesting bugs, but most of the time becomes tiring to the programmer. The verifier VeriCool [39] tries to infer such statements automatically, but, in the author's opinion, for most interesting examples this is a burden and not a help for the programmer.

6 Related Work

The frame problem [29] has been recognized as one of the hardest issues for the specification of object-oriented programming since the late nineties [10]. Previous attempts to deal with the problem include *Data Groups* [21], as well as Leino and Nelson's methodology [28]. Data Groups resembles Dynamic Frames, but is less expressive. The methodology of [28] is complicated and only deals with the problem partially.

A school of thought in the area is the use of *Ownership* [7]. Ownership-based frameworks are used to describe heap topologies and policies of mutation, among other things. They can thus be used to address the framing problem. The *Universes* type system [31] is designed for this purpose, and has been used in Spec# [25, 2], one of the strongest and most comprehensive verification languages up to today. VCC [8] is another popular tool based on ownership. Ownership based solutions tend to be very simple and intuitive, but they have typically problems with patterns that use *sharing*, such as the *Iterator* pattern [13] (for example, the Spec# solution for sharing [3] has modularity issues [17]). Research on Ownership is currently very active.

Separation Logic [35] started off as a low level extension of Hoare-logic with a new operator, that of separating conjunction. With the invention of Abstract Predicates [32], it became a mainstream solution to the frame problem. Most Separation Logic-based theories are not oriented to automation. Those that are, typically reduce the specification language to a small subset of the original (for example they throw away non-separating conjunction, just like Implicit Dynamic Frames) and then apply Symbolic Execution [20]. Such tools are

[16, 33, 5]. The semantic relationship of the separation logics supported by these tools to Implicit Dynamic Frames is explored in a recent paper [34].

Besides the ideas presented here, the automation of verification of Dynamic Frames is a focus of Regional Logic [1], which uses ghost fields in a way similar to Dafny. Dynamic Frames is also used in the new version of the KeY verifier [36, 4], a partially automated symbolic execution engine. The verification there may be done using ghost fields, but fully second-order inductive definitions are also allowed, since the option of interaction is open.

Implicit Dynamic Frames is a variant of Dynamic Frames introduced in [38], together with VeriCool, the first tool that supported this style of specification. The language Chalice [24], which we used in this tutorial, extended Implicit Dynamic Frames with Fractional Permissions [6]. Chalice can be used to verify absence of deadlocks and data races, as well as functional properties, in lock-based concurrent programs, using thread modularity. Chalice was also extended with modest support for message passing [27]. The concurrency features of Chalice are not presented in this tutorial. An instructive tutorial is [26]. The first verifier that came with Chalice is a VCG tool based on Boogie. Syxc [37] is a newly developed SE tool.

7 Conclusion

In this tutorial, we have presented the formalism of Dynamic Frames, and we outlined two approaches that can be used to automate the verification of programs that are specified using this formalism. It is hoped that the information collected here can motivate and support further research on the problem.

References

[1] A. Banerjee, D. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP'08*, volume 5142 of *Lecture Notes In Computer Science*, pages 387–411. Springer-Verlag, 2008.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# specification language: an overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS'04*, volume 3362 of *Lecture Notes In Computer Science*, pages 49–69. Springer-Verlag, 2004.

[3] M. Barnett and D. Naumann. Friends need a bit more: maintaining invariants over shared state. In D. Kozen, editor, *MPC'04*, volume 3125 of *Lecture Notes In Computer Science*, pages 54–84. Springer-Verlag, 2004.

[4] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes In Computer Science*. Springer-Verlag, 2007.

[5] J. Berdine, Cristiano Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO'05*, volume 4111 of *Lecture Notes In Computer Science*, pages 115–137. Springer-Verlag, 2006.

[6] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *SA'03*, volume 2694 of *Lecture Notes In Computer Science*, pages 55–72. Springer-Verlag, 2003.

[7] D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, 2001.

[8] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskał, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes In Computer Science*, 2009.

[9] Detlefs D, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[10] D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep-exposure. Technical Report 156, DEC-SRC, 1998.

[11] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall Series in Automatic Computation. Prentice Hall, 1976.

[12] J. C. Filliâtre. Why: a multi-language multi-prover verification tool. Technical Report 1366, LRI, Université Paris Sud, 2003.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[14] E. C. R. Hehner. Termination is timing. In J. L. A. van de Snepscheut, editor, *MPC'89*, volume 375 of *Lecture Notes In Computer Science*, pages 36–47. Springer-Verlag, 1989.

[15] C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.

[16] B. Jacobs, J. Smans, and F. Piessens. VeriFast: Imperative programs as proofs. In *VS-Tools workshop at VSTTE'10*, 2010.

[17] I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM'06*, volume 4085 of *Lecture Notes In Computer Science*, pages 268–283. Springer-Verlag, 2006.

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

[19] I. T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23(3):267–289, 2011.

[20] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[21] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA'98*, pages 144–153. ACM, 1998.

[22] K. R. M. Leino. Specification and verification of object-oriented software. In *Marktoberdorf International Summer School 2008, Lecture Notes*, 2008.

[23] K. R. M. Leino. This is Boogie 2. Working Draft - available at `http://-research.microsoft.com/en-us/um/people/leino/papers.html`, 2008.

[24] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *ESOP'09*, volume 5502 of *Lecture Notes In Computer Science*, pages 378–393. Springer-Verlag, 2009.

[25] K. R. M. Leino and P. Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In P. Müller, editor, *Advanced Lectures on Software Engineering—LASER Summer School 2007/2008*, volume 6029 of *Lecture Notes In Computer Science*, pages 91–139. Springer-Verlag, 2010.

[26] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes In Computer Science*, pages 195–222. Springer-Verlag, 2009.

[27] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In A. D. Gordon, editor, *ESOP'10*, volume 6012 of *Lecture Notes In Computer Science*, pages 407–426. Springer-Verlag, 2010.

[28] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5), 2002.

[29] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[30] L. Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS'08*, volume 4963 of *Lecture Notes In Computer Science*, pages 337–340. Springer-Verlag, 2008.

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

[32] M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL'05*, pages 247–258, 2005.

[33] M. Parkinson and D. Distefano. jStar: Towards practical verification for Java. In G. E. Harris, editor, *OOPSLA'08*, pages 213–226. ACM, 2008.

[34] M. Parkinson and A. Summers. The relationship between separation logic and implicit dynamic frames. In Gilles Barthe, editor, *ESOP'11*, volume 6602 of *Lecture Notes In Computer Science*. Springer-Verlag, 2011.

[35] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE Computer Society, 2002.

[36] P. Schmitt, M. Ulbrich, and B. Weiß. Dynamic frames in Java dynamic logic. In *FoVeOOS'10*, volume 6528 of *Lecture Notes In Computer Science*, pages 138–152. Springer-Verlag, 2011.

[37] M. Schwerhoff. Symbolic execution for Chalice. MSc thesis Dept. of Computer Science, ETH Zurich, 2011.

[38] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP'09*, Genoa, pages 148–172. Springer-Verlag, 2009.

[39] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In S. Drossopoulou, editor, *ECOOP'09*, volume 5653 of *Lecture Notes In Computer Science*, pages 148–172. Springer-Verlag, 2009.