Automated Stepwise Refinement of Heap-Manipulating Code

Manuscript KRML 211– October 9, 2010

K. Rustan M. Leino[0] and Kuat Yessenov[1] ⋆

[0] Microsoft Research, Redmond, CA, USA
leino@microsoft.com
[1] MIT Computer Science and Artificial Intelligence Lab, Cambridge, MA, USA
kuat@csail.mit.edu

Abstract. Stepwise refinement is a well-studied technique for developing a program from an abstract description to a concrete implementation. This paper describes a system with automated tool support for refinement, powered by a state-of-the-art verification engine that uses an SMT solver. Unlike previous refinement systems, users of the presented system interact only via declarations in the programming language. Another novel aspect of the system is that it accounts for dynamically allocated objects in the heap, so that data representations in an abstract program can be refined into ones that use more objects. Finally, the system uses a language with familiar object-oriented features, including sequential composition, loops, and recursive calls, offers a syntax with skeletons for describing program changes between refinements, and provides a mechanism for providing witnesses when using angelic non-determinism.

0 Introduction

The prevalent style of programming today uses low-level programming languages (like C or Java) into which programmers encode the high-level design or informal specifications they have in mind. From a historical perspective, it makes sense that this style would have come from the view that what the programming language provides is a description of the data structures and code that the executing program will use. However, upon reflection, the style seems far from ideal, for several reasons. First, the gap between informal specifications to executable code is unnecessarily large, leaving much room for errors. Second, errors in the informal specifications may best be discovered by execution, simulation, or property discovery, but such processes cannot be applied until a machine readable description—here, the low-level code—is in place. Third, programmers often understand algorithms in terms of pseudo-code, which abstracts over many nitty-gritty details, but such pseudo-code is confined to whiteboards or the heads of programmers, rather than being recorded as part of the program text. Fourth, interesting software goes through considerable evolution, which includes the introduction of various optimizations; these usually take the place of the old code, making them harder

⋆ work done during an internship at Microsoft Research

to understand both when they are being developed ("is this really doing what the unoptimized code did?") and when the code is later examined for human understanding. Fifth, another important fact of program evolution is that it involves multiple developers, whose introduction to the code immediately takes them into the gory depths of all the low-level decisions that have been made.

An alternative style of programming uses *stepwise refinement*, starting from a higher-level description of what the program is intended to do and then gives various levels of pseudo-code until the low-level code is in place. This is an old idea due to Dijkstra [20] and Wirth [44] and given mathematical rigor by Back [5]. It underwent much theoretical development during the 1980's and 1990's (*e.g.*, [8, 39, 40, 24, 27, 0]). The technique has been successfully applied in practice where program correctness has been critical (*e.g.*, [2, 16, 36]). Some tool sets, like Rodin [4] and Atelier B [16], support the refinement process.

To reap the benefits of the refinement process, the intermediate stages of program development (that is, the various levels of pseudo-code) must be recorded and preserved in a format that is appropriate for consumption by human engineers as well as analysis tools. In computer science, we usually refer to such a format as a programming language (or modeling language, or specification language). As they work with it, the language and its associated tool set become the engineers' primary thinking aid.

In this paper, we take refinement closer to important facilities of present-day programming and verification.

On the programming side, we use an object-oriented language, which means that the various stages of refinement look more like the code programmers are used to writing. The implementation of a class is often built on other (tailor-made or library-provided) classes. More precisely, the data of an object is represented by the object's fields and by other dynamically created objects accessible from those fields. While this is taken for granted by programmers, we are not aware of any previous treatment of refinement that allows abstract fields to be refined into new objects of instantiable classes.

On the verification side, we integrate automatic verification support, like that found in leading-edge program verifiers (*e.g.*, [17, 12, 9, 10, 29]), based on a satisfiability-modulo-theories (SMT)-solver foundation. This means that programmers can focus more on the program under development with fewer distractions of having to manually guide a separate proof assistant.

More specifically, our contributions in this paper are:

0. a foundation, based on a model of memory permissions, that works for heap-manipulating code and allows refinement steps to introduce new object instances in data representations
1. a checking algorithm that encodes refinement proof obligations (as input to an automatic verification engine) to harness the power and automation provided by an SMT solver
2. facilities in the language for describing a refinement in terms of the differences from the previous refinement and for supplying an abstract witness when coupling relations are non-deterministic
3. a prototype implementation as an extension of the language and verifier Chalice [31, 32], which uses the Boogie verification engine [34] and the Z3 SMT solver [19].

In Sec. 1, we review refinement, using an example in our refinement system. We then describe a problem that arises when trying to introduce instances of a reusable class as part of the data representation of another object. In Sec. 2, we review the model of memory permissions in Chalice and then present how we use that model to provide a sound solution to the data-refinement problem. We describe our checking algorithm in Sec. 3 and our syntactic *skeletons* facility in Sec. 4.

1 Introductory Examples

Intuitively, to say that a program A is *refined by* a program B is to say that for any context where A's behavior is acceptable, substituting B for A would also make for acceptable behavior. B's behavior is acceptable wherever A's behavior is. We take the behavior of a program to be what can be observed by relating its possible pre- and post-states. In our setting, a class is refined by another if all its methods are refined by the corresponding methods of the other class. Consequently, the compiler or user can freely choose to replace a class by one of its refinements, while maintaining the correctness of the program.

In Sec. 1.0, we review refinement by walking through an example development of a program in our system. The refinement steps will be familiar to anyone acquainted with stepwise refinement; the example gives us the opportunity to showcase how one works with our system. The example is also available in video form as an episode of Verification Corner[0]. In Sec. 1.1, we describe a problem with data refinement and objects.

1.0 Algorithmic Refinement

Top-Level Description Let us write a procedure that computes whether or not a given sequence has any duplicated elements. We introduce the procedure as a method in a class, as one would in an object-oriented language.

The initial description of the behavior of this method can be given as a pre- and post-condition specification à la Eiffel [37], the precondition describing when the method is defined and the postcondition describing its effect. However, there are cases where it is more straightforward to describe the effect using a method body. In Fig. 0, we use the latter option (with a trivial, and hence omitted, precondition `true`).

A sequence in our language is a mathematical value, just like booleans and integers. A sequence subscripted by a single index returns that element of the sequence; subscripted by an interval, it returns the subsequence consisting of the specified elements. Sequence indices start with 0, the length of a sequence `s` is denoted `|s|`, and `s[i := s]` (used later) denotes a sequence like `s` except that element `i` has the value `e`. Every interval `[a..b]` is half-open, that is, it denotes the integers x that satisfy $\mathtt{a} \leqslant x < \mathtt{b}$. The existential quantifier in the specification statement in Fig. 0 can be read as "there exists an index `i` in the range from 0 to less than the length of `s`, such that element `i` of `s` also occurs among the first `i` elements of `s`". In other words, the existential evaluates to `true` iff `s` has a duplicate element.

[0] `http://research.microsoft.com/verificationcorner`

```
class Duplicates0 {
  method Find(s: seq<int>) returns (b: bool)
  {
    b := exists i in [0..|s|] :: s[i] in s[0..i];
  }
}
```

Fig. 0: An initial description of a method that checks for duplicate elements in given sequence. The method `Find` has an in-parameter `s` and an out-parameter `b`.

```
class Duplicates1 refines Duplicates0 {
  refines Find(s: seq<int>) returns (b: bool)
  {
    var n := 0;
    b := false;
    while (n < |s|)
      invariant 0 ⩽ n ∧ n ⩽ |s|;
      invariant b <==> exists i in [0..n] :: s[i] in s[0..i];
    {
      var c := s[n] in s[0..n];
      b := b ∨ c;
      n := n + 1;
    }
  }
}
```

Fig. 1: A refined `Find` method, where the specification statement in Fig. 0 has been replaced by code that uses a loop.

Because this is the initial description of our method, there is nothing to verify, other than the well-definedness of the operations used. In particular, there is no check that this actually describes the program we have in mind. However, since this description is clearer than, say, an optimized program with loops, a human may stand a better chance of proof reading this description.

In summary, the thing to notice about our program's initial description in Fig. 0 is the emphasis on what is to be computed, not how it is computed.

Introducing a Loop A compiler may or may not be able to compile the existential quantifier we used in the body of `Find`, and it is unlikely to compile it efficiently. So, let's help it along. Figure 1 introduces a class whose `Find` method refines the one in Fig. 0. To reason about the loop, we supply a loop invariant; our system checks the invariant to hold on entry to the loop and to be maintained by the loop body. The loop invariant and the negation of the loop guard imply that `b` will end with the same value as in Fig. 0, hence establishing the correctness of the refinement.

The transformation from Fig. 0 to Fig. 1 offers two key benefits to programmers. First, both versions of the program remain part of the program text. This means that someone trying to understand the program can start by studying the more abstract description in Fig. 0 and then move to the more concrete description in Fig. 1. Second, our system verifies the correctness of the transformation (in less than 0.05 seconds). This checks the refinement step to be correct; furthermore, it ensures that future changes to either Fig. 0 or Fig. 1 will keep the two in synch. The proof does not come entirely for free, since loop invariants have to be supplied by the user, but in contrast to previous refinement tools, the interaction stays at the level of the program and the user never issues any commands to the underlying theorem prover.

Adding an Efficient Data Structure The method in Fig. 1 still contains a point of inefficiency, namely the assignment to `c`. Let's do another refinement, this time adding (in the jargon, *superimposing*) a sequence of booleans that keeps track of which numbers have been encountered so far.

To simplify matters, we will add a restriction to our original program, limiting the elements of `s` to be among the first 100 natural numbers, as expressed by the following precondition:

```
requires forall i in s :: i in [0..100];        (0)
```

This going back to and changing the original description is common in practice, because all necessary restrictions may not be evident at the onset of the program development [2].

Figure 2 shows the new refinement. It uses the keyword **`transforms`** for method `Find`, which allows us to transform the method body at the level of its statements. (The keyword **`refines`** we used in Fig. 1 is a special case of **`transforms`** that says the entire method body is being replaced.) The body of `Find` in Fig. 2 uses a *skeleton* syntax that we will describe in Sec. 4. Essentially, a skeleton keeps the structure of **`if`** and **`while`** statements (but does not syntactically repeat guards or invariants), has the option of replacing (keyword **`replaces`**) various update statements, can add (superimpose) new statements, and uses "_" as a wildcard denoting other statement sequences of the method body being transformed.

Our refinement introduces `bitset` as a sequence of 100 booleans, all initially **`false`** (*i.e.*, initially, **`true`** is not in the sequence, which we conveniently express here using a *specification statement* [39]). The loop body sets element `s[i]` of `bitset` to **`true`**, thus maintaining the properties that are recorded as loop invariants: the length of `bitset` remains 100, any element `s[i]` encountered so far has been recorded in `bitset`, and anything recorded in `bitset` has been encountered in `s`.

With these properties of `bitset`, we are able to replace the assignment of `c` with a simpler assignment statement. When the refinement in Fig. 2 is verified, the loop invariants in Fig. 1 do not need to be re-verified and neither does the postcondition that was verified in Fig. 1. In this way, refinement localizes proof obligations.

Summarizing the Example This concludes our introductory example. One can imagine further refinements, such as changing `bitset` from being a sequence to being an

```
class Duplicates2 refines Duplicates1 {
  transforms Find(s: seq<int>) returns (b: bool)
  {
    _;
    var bitset: seq<bool> [|bitset| == 100 ∧ true !in bitset];
    while
      invariant |bitset| == 100;
      invariant forall j in [0..100] :: bitset[j] <==> j in s[0..n];
    {
      replaces c by {
        var c := bitset[ s[n] ];
      }
      bitset := bitset[s[n] := true];
      _;
    }
  }
}
```

Fig. 2: A further refinement of `Find`, introducing a sequence of booleans that keep track of which numbers have been encountered so far by the loop. The correctness of the code relies on including precondition (0) in the original description of `Find` in Fig. 0.

array (to avoid the costly sequence-update operation in the loop in Fig. 2, or terminating the loop as soon as `b` is set to **`true`**, or avoiding the loop altogether if the length of `s` exceeds 100.

Given Figs. 0, 1, and 2 and the precondition (0), our system performs the verification automatically in about 1 second.

1.1 Data Refinement

The previous example did not involve the heap. Our next example does. We review the idea of data refinement and demonstrate an important problem that has been insufficiently addressed in the literature.

Our motivating example comes in three pieces: a class, a client of the class, and a refinement of the class. If a sound refinement system verifies these pieces, then one can replace the client's use of the class by the refined class. In our example, such a replacement would lead to a run-time error, which tells us that soundness requires the refinement system to report some error. The question is then *where* the error is to be detected and reported during verification.

The class we consider is a simple counter, see Fig. 3. Method `Get()` returns the current value of the counter and `Inc()` increments it. The somewhat mysterious method `M()` is described as returning any `Cell` object, where `Cell` is another class shown in the figure. The specification statement in the body of `M()` says to set `r` to any value satisfying the condition in brackets. It seems reasonable that a verification system would consider classes `Counter` and `Cell` to be correct.

```
class Counter {
  var n: int;
  method Get() returns (r: int) { r := n; }
  method Inc() { n := n + 1; }
  method M() returns (r: Cell) { spec r [true]; }
}
class Cell {
  var x: int;
}
```

Fig. 3: A simple class that provides the functionality of a counter, as well as (a rather unmotivated) method that returns a cell object.

In Fig. 4, we show a client of the `Counter` class. It allocates a `Counter`. Then, it calls `Get()` twice and checks, with an **`assert`** statement, that the counter was unchanged. Between the two calls to `Get()`, it obtains a `Cell` via the `M()` method and sets the cell's `x` field to the arbitrary value 12.[1] Here is one way one might argue for the correctness of the client: the description of `M()` says that the only effect of `M()` is to set its out-parameter, updating `cell.x` has no effect on `cnt.n` (after all, `x` and `n` are different fields and `cell` and `cnt` are not aliased since they point to objects of different types), and therefore the correctness of the assert follows from the description of `Get()`.

```
class Client {
  method Main() {
    var cnt := new Counter;
    call a := cnt.Get();
    call cell := cnt.M();
    cell.x := 12;
    call b := cnt.Get();
    assert a == b;
  }
}
```

Fig. 4: An example client of the code in Fig. 3. This code is correct only if the asserted condition will always evaluates to **`true`**.

In Fig. 5, we show a refinement of class `Counter`. It superimposes a field `c` into which it dynamically allocates a new `Cell` object and maintains the *coupling invariant* `n == c.x`. This kind of *data refinement*, where one data representation is replaced by another, has been studied extensively (*e.g.*, [26, 39, 24]), but—surprisingly–not in the

[1] If the direct access of field `x` in class `Client` bothers you, you may consider our same but with a `SetX` method in `Cell`.

presence of pointers and dynamic storage! In our example, which uses pointers and dynamic storage, one might argue that `CCounter` is a correct refinement of `Counter` as follows (for now, we ignore some issues, like initialization): Whatever `Get()` and `Inc()` did with `n` in `Counter`, they now do with `c.x` in `CCounter`; moreover, `Counter` says nothing about what `Cell` is returned by `M()`, giving `CCounter` total freedom in what it returns.

```
class CCounter refines Counter {
  var c := new Cell;
  refines Get() returns (r: int) { r := c.x; }
  refines Inc() { c.x := c.x + 1; }
  refines M() returns (r: Cell) { r := c; }
}
```

Fig. 5: A sketch of a class to refine the behavior of `Counter` in Fig. 3. Class `CCounter` implements `n` in `Counter` by `c.x`.

2 Heap Refinement

The memory model that underlies our heap-aware refinements uses permissions [13] and implicit dynamic frames [42]. The model forms a core of the language and verifier Chalice [31, 32], into which we have incorporated our refinement system. Chalice and our extensions are available as open source[2] and can be run either from the command line or from within the Microsoft Visual Studio IDE.

2.0 Permissions

A heap location is identified by an object-field pair. Heap locations have associated access permissions, which can be transferred between activation records (*i.e.*, method-invocation instances and loop iterations) in a running program. Every heap-location access (*i.e.*, read or write) requires the current activation record to have sufficient permissions for the access. Permissions are *ghost* entities: they can be mentioned in specifications and are used by the verifier, but they need not be present at run-time in a verified program.

For example, the `Inc` method in Fig. 3 reads and writes the field `n`. As shown in the figure, the Chalice verifier will report an error of insufficient permissions for these accesses, because activation records of `Inc()` have no permissions. To equip `Inc()` with permission to access `n`, one declares a precondition `requires acc(n);`. The evaluation of this precondition checks that the caller does indeed have access to `n` and then transfers that permission to the callee. In this example, it is also desirable to return the permission to the caller, which is achieved by declaring a postcondition `ensures acc(n);`.

[2] `http://boogie.codeplex.com`

Specifications can mention several access predicates, which are evaluated in order. For example, suppose a method declares the precondition `acc(x) ∧ acc(y)`. The caller will then be checked for permission to `x`, then that permission will be transferred to the callee, then the permission to `y` will be checked and transferred.

Permissions can be divided among activation records. Write access requires full permission (100%), whereas any non-zero fraction of the full permission suffices for read access. Syntactically, a fractional permission is indicated by supplying a second argument to `acc`, specifying a percentage of the full permission; for example, `acc(x,50)` indicates half of the permission to `x`.

If, after evaluating the precondition, a caller still has some permission to a heap location, then the caller can be sure the callee will not modify the heap location because the callee will not be able to obtain the full permission. Because of the evaluation order of predicates, `acc(x,50) ∧ acc(x,50)` is equivalent to `acc(x)`, since the two fractions add up to the full permission; and the condition `acc(x,80) ∧ acc(x,30)` is never satisfiable, since 110% is more than 100%.

Note that all proper fractions grant the same permission to read; 1% and 20% and 99% are all the same in this respect. The reason for keeping track of specific fractions is so that one can determine if various fractions add up to 100%, which would imply write permission.

When an activation records allocates a new object, it receives full permission to all fields of the object. It is possible for a program to squander permissions: any permission remaining in an activation record after the postcondition has been evaluated is forever lost, in effect rendering the corresponding heap locations readonly.

Access predicates can only be mentioned in positive positions (*e.g.*, not as antecedents of implications). For more details about permissions in Chalice, see [31].

Consider the `Counter` example in Sec. 1.1. One way to make it verify is to declare `acc(n)` as a pre- and postcondition of `Inc()` and `Get()`. (Alternatively, `Get()` could use a fractional permission, since it only reads `n`.) This would also verify the client in Fig. 4, if it were not for the update of `cell.x`, for which the client has no permissions. As it stands, method `M()` says nothing about the `Cell` being returned. We can change `M()` to say that it will also return full permission to the `x` field of the returned object:

```
method M() returns (r: Cell)
  ensures acc(r.x);
{ spec r [acc(r.x)]; }
```

Now, both `Counter` and `Client` verify.

2.1 Coupling Invariants for the Heap

Coupling invariants (or abstraction functions and relations) are critical in establishing data refinement. Traditionally, they are logical formulas that relate concrete variables to abstract variables. In Chalice, coupling invariants relate abstract heap locations to superimposed concrete heap locations. For class `CCounter` (Fig. 5), one would want to relate field `n` of `Counter` and field `c` of `CCounter` via the following declaration:

```
replaces n by acc(c) ∧ acc(c.x) ∧ n == c.x        (1)
```

While the latter part of the formula is the familiar logical formula, the former part is unique to Chalice's permission system. Intuitively, this coupling invariant gives `CCounter` a license to trade permissions to access `n` for permissions to access `c` and `c.x`, appropriately scaled. Given such a license, the body of method `Inc` may write the field `c.x`, since it has full access to `n` that by virute of the coupling invariants warrants full access to `c.x`.

The general form of the coupling invariant declaration permits simultaneous replacement of several abstracted heap locations:

replaces $f_1, \ldots, f_k$ **by** `I`

The access predicates in the coupling invariant `I` are then split evenly between f_i. This rule eliminates a possibility that a write to concrete representation occurs while another activation record holds read access to abstract representation $\{f_i\}$.

Going back to the `CCounter` example, refinement `M` fails to verify since assignment `x := c` has insufficient permissions to read `c`. Now imagine that we add precondition **acc**`(n)` to `CCounter.M`. We should also add postcondition **acc**`(n)` or, otherwise, the client is not able to inspect `n` after making a call to `M`. But even now, method `M` fails to verify. The reason is that at the end of its execution, both **acc**`(n)` and **acc**`(c.x)` imply full access to `c.x`. Therefore, the postcondition is never satisfiable and refinement `CCounter` fails to verify.

3 Checking Algorithm

We want to leverage the power of an automatic reasoning engine, like the collection of first-order decision procedures available in modern satisfiability-modulo-theories (SMT) solvers (*e.g.*, [19]). How to produce input for such a reasoning engine is well known (see, *e.g.*, [10]): essentially, one produces a formula of the form

$$P \;\Rightarrow\; \mathsf{wp}[\![B, Q]\!] \tag{2}$$

where P and Q are the declared pre- and postconditions of a procedure, B is the body of that procedure, and $\mathsf{wp}[\![B, Q]\!]$ is the *weakest precondition* of B with respect to Q [21]. If expressions are first-order terms and loops and calls are handled via specifications (as usual), then (2) will be a first-order formula. However, to verify that a program B refines a program A, one needs to check that B can be substituted for A in any context, which is expressed in terms of weakest preconditions as

$$(\,\forall Q \bullet\; \mathsf{wp}[\![A, Q]\!] \;\Rightarrow\; \mathsf{wp}[\![B, Q]\!]\,) \tag{3}$$

where the quantification of Q ranges over any predicates [5]. Since this is a second-order formula, it is not directly suitable as input to an SMT solver.

To express formula (3) in a first-order setting, we apply two techniques. First, monotonicity of the refinement relation with respect to the sequential composition permits us to prove it locally for isolated statements and blocks of code. A block in the abstract program is matched against its *refinement block* in the concrete program. Second,

non-deterministic abstract statements are refined separately by refinement blocks that produce witnesses to such statements.

Refinement condition (3) can be expressed in a different form that avoids predicate quantification using *coupling invariant* I [23]:

$$\mathsf{wp}[\![A, \top]\!] \wedge I \;\Rightarrow\; \mathsf{wp}[\![B, \neg \mathsf{wp}[\![A, \neg I]\!]]\!] \tag{4}$$

In other words, for any execution of B (starting from an initial state satisfying I and on which A is defined), there is a possible *angelic* execution of A such that I is reestablished in the final states of B and A. If A is deterministic, then any execution is angelic and we can cancel double negation in the formula (4), and simplify it down to:

$$\mathsf{wp}[\![A, \top]\!] \;\Rightarrow\; \mathsf{wp}[\![\texttt{assume}\ I; B; A; \texttt{assert}\ I, \top]\!] \tag{5}$$

Here A and B operate in disjoint state spaces but their initial and final states are paired using I. We have already mentioned how I can be declared for superimposed heap locations using **replaces** keyword. Local variables of A are bound to local variables of B by simple equality. Superimposed local variables in program B are left unconstrained by I.

If program A is non-deterministic then formula (5) is a sound but not a complete characterization of refinement. Chalice provides two ways to introduce non-determinism into a program: specification statements and call statements. Both are specified using declarative pre- and postconditions. Verifying refinement of a single non-deterministic statement A by a program B then amounts to extracting witnesses from B that satisfy the postcondition of A.

In Chalice, programs are structured into classes and methods. To verify that method m in class A is refined by method m in class B, we check that:

0. $B.m$ has the same pre-condition but possibly stronger post-condition.
1. $B.m$ accepts same inputs as $A.m$ and returns as many outputs plus possibly more.
2. The body of $B.m$ is a refinement of the body of $A.m$.

The surface syntax (see Sec. 4) allows us to identify correspondence between abstract statements of $A.m$ and concrete statements of $B.m$. Once we localize code substitutions to disjoint refinement blocks $R[P, Q]$ and loop refinements $L[I, P]$, we can generate a Boogie program C that encodes the refinement condition [34]. Here P is a block of code within the body of $A.m$, Q is the replacement block of code in the refinement $B.m$, and I are new loop invariants in $B.m$. Boogie program C is constructed from a Boogie translation of $A.m$ as follows:

Declaring superimposed state: Program C takes same inputs as $A.m$ and produces same outputs as $B.m$. The super-imposed local variables and fields of B are also declared in C.

Sequential refinement block $R[P, Q]$*:* A sequence of statements P in $A.m$ that is a part of a refinement block is transformed into the following sequence of instructions in Boogie intermediate language:

0. Create a duplicate state. The state consists of the heap, the permission mask, and local variables.
1. Permissions to access superimposed fields are derived from permissions of the abstract heap locations using the coupling invariant. These permissions are inhaled into the secondary copy along with the coupling invariant. Permissions to access replaced fields are exhaled from the secondary copy.
2. Execute P from A using the primary copy of the state. Execute Q from B using the secondary copy of the state.
3. Assert coupling invariant between the two copies of the state. For local variables, it amounts to simple equality. Superimposed variables and their permissions are carried over to the primary state. For every replaced field, its coupling invariant is asserted. We also check that the secondary copy holds enough permissions to superimposed variables replacing the field.

Loop refinement $L[I, P]$*:* We add assertions to establish the new loop invariant I at the entrance of the loop and to show that the body P maintains it. The body of the loop itself might contain refinement blocks and loop refinements.

Refinement of a non-deterministic statement $R[P, Q]$*:* If P is a single non-deterministic statement, then we replace P with

$$Q; \texttt{assert}\ post[P]$$

where $post[P]$ is the post-condition of P.

Assume correctness of the abstract program: Correctness of the abstract program is a strong hypothesis that allows us to eliminate pre-existing assertions of A in C. We change every assertion that was present in A (including loop invariants) into an assumption. Refinement of B is conditional on refinement conditions of A and its previous refinement steps as well as the correctness guarantees of the top-level abstraction.

Example Figure 6 shows two programs for computing a sum of cubes. Our system translates the abstract program into an intermediate language Boogie ([34]) and applies the transformation. A snippet of the resulting Boogie program is shown in Fig. 6c. The final Boogie program is then fed to an automated theorem prover Z3 [19].

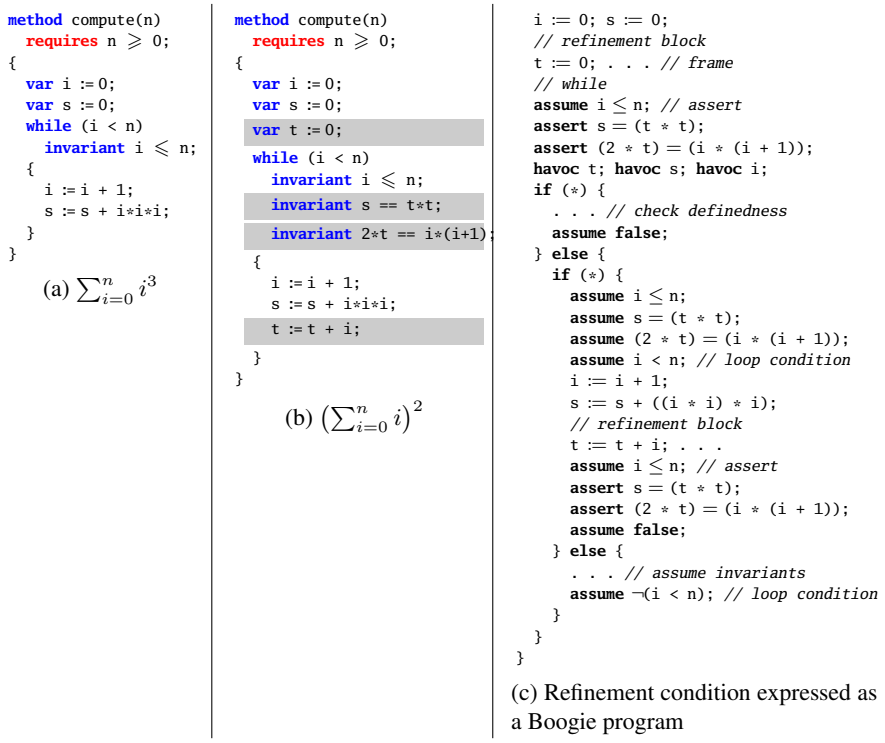
4 Surface Syntax

In this section, we present our extensions to the syntax of Chalice [32].

4.0 Class refinement

We extended the syntax of Chalice with a declaration for class refinement:

```
class B refines A { ... }
```



```
method compute(n)
  requires n ⩾ 0;
{
  var i := 0;
  var s := 0;
  while (i < n)
    invariant i ⩽ n;
  {
    i := i + 1;
    s := s + i*i*i;
  }
}
```

(a) $\sum_{i=0}^{n} i^3$

```
method compute(n)
  requires n ⩾ 0;
{
  var i := 0;
  var s := 0;
  var t := 0;
  while (i < n)
    invariant i ⩽ n;
    invariant s == t*t;
    invariant 2*t == i*(i+1);
  {
    i := i + 1;
    s := s + i*i*i;
    t := t + i;
  }
}
```

(b) $\left(\sum_{i=0}^{n} i\right)^2$

```
i := 0; s := 0;
// refinement block
t := 0; . . . // frame
// while
assume i ≤ n; // assert
assert s = (t * t);
assert (2 * t) = (i * (i + 1));
havoc t; havoc s; havoc i;
if (*) {
  . . . // check definedness
  assume false;
} else {
  if (*) {
    assume i ≤ n;
    assume s = (t * t);
    assume (2 * t) = (i * (i + 1));
    assume i < n; // loop condition
    i := i + 1;
    s := s + ((i * i) * i);
    // refinement block
    t := t + i; . . .
    assume i ≤ n; // assert
    assert s = (t * t);
    assert (2 * t) = (i * (i + 1));
    assume false;
  } else {
    . . . // assume invariants
    assume ¬(i < n); // loop condition
  }
 }
}
```

(c) Refinement condition expressed as a Boogie program

Fig. 6: Refinement of a program for computing sum of cubes and its simplified encoding into Boogie. The highlighted lines show the new code in the concrete program.

This declaration introduces class B as a refinement of class A. We refer to B as a concrete class and to A as an abstract class. For B to be a valid refinement of A, it must satisfy the following three conditions:

0. Every declared member of A is present in B. B may *refine* a subset of methods of A but the rest are carried over to B. Similarly, fields of A are also fields of B (however, some fields may turn ghost via data refinement.) B may declare new methods and fields. We call the latter *superimposed* fields.
1. B may declare a method m to be a refinement of a method m in A using either **`refines`** or **`transforms`** keywords instead of **`method`** declaration. We describe the difference between these two types of declarations in the next paragraph.
2. B may declare a global *coupling invariant* using the following declaration:

```
replaces x by acc(y) ∧ acc(z) ∧ x == y - z
```

In this example, field `x` of A is bound to superimposed fields `y` and `z` of B by means of an abstraction relation `x == y - z`.

More often that not, individual methods of a concrete class require only a small number of changes to select statements of the corresponding method of the abstract

class. The programmer's insight to deriving such a concrete, refined implementation can often be expressed as a set of transformation rules that introduce new statements and substitute parts of the abstract program. Example 6 demonstrates one such scenario: the real insight behind this refinement is the mathematical identify $\sum_{i=0}^{n} i^3 = (\sum_{i=0}^{n} i)^2$ that lets one compute a sum of cubes with just one multiplication. To implement this optimization, a programmer needs to introduce a new local variable `t` and establish coupling with variable `s` using a loop invariant. This transformation can be succinctly expressed in Chalice as a *skeleton*:

```
transforms compute(n)
{
  _
  var t := 0;
  while
    invariant s == t*t;
    invariant 2*t == i*(i+1);
  {
    _;
    t := t + 1;
  }
}
```

Fig. 7: An example of a skeleton for the program in Fig. 6

Skeleton methods such as the one in Fig. 6 are declared using keyword **`transforms`**. Fig. 1 shows another way to declare a refined method. **`refines`** keyword is used to mark a method that substitutes the entire code in the abstract method by concrete code supplied by the declaration.

4.1 Skeletons

Skeletons are transformation rules that are composed of code navigation and rewrite operations. Given an abstract program, skeleton serves as a template that is filled in by statements taken from the abstract program. It does so by pattern matching control flow of the abstract program against a set of pre-defined primitive substitutions.

Abstractly, skeleton is a partial function from an abstract syntax tree (AST) of an abstract program to an AST of the concrete program. Since skeleton maintains the original control flow structure, it helps us to think of the resulting program as consisting of normal statements, *refinement blocks*, where each such block is a pair $R[A, B]$ of an abstract statement sequence A and concrete statement sequence B, and *loop refinements* $L[I, P]$, which add loop invariant I to an existing loop and replace its body by P. In a well formed refinement block $R[A, B]$, B declares all the local variables declared in A. Our checking algorithm takes full advantage of the fine structural mapping between the abstract and concrete code embodied in these refinement blocks and loop refinements.

A skeleton $\mathcal{S}$ is defined inductively from a set of primitive wild card skeletons and sequential composition:

- Skeleton _ is a block pattern that matches any sequence of non-conditional deterministic statements and acts as identity.
- Skeleton $*$ matches any sequence of statements and acts as identity.
- Skeleton **replaces** $*$ **by** { B } matches any sequence of statements A and produces $R[A, B]$.
- Skeleton **if** { $\mathcal{S}_1$ } matches a single **if** statement and produces an **if** statement with $\mathcal{S}_1$ applied to its branch; skeleton **if** {$\mathcal{S}_1$ } **else** { $\mathcal{S}_2$ } is analogous.
- Skeleton **while invariant** I { $\mathcal{S}_1$ } matches a single **while** loop and produces a **while** loop with an additional loop invariant I and the body P that is obtained by applying skeleton $\mathcal{S}_1$ to the body of the original loop. We use notation $L[I, P]$ for such loop refinements.
- Skeleton **replaces** v **by** { B } matches any statement A that effects variables in list v. The resulting refinement block is $R[A, B]$. This pattern is used to provide witnesses to non-deterministic specifications or call statements and also rewrite assignment statements. Our checking algorithm resolves angelic non-determinism as described in Sec. 3.
- Skeleton B consisting of Chalice statements matches only the empty program and produces $R[\emptyset, B]$.
- Sequential skeleton $\mathcal{S}_1$; $\mathcal{S}_2$ matches $\mathcal{S}_1$ greedily (*i.e.* consuming as many statements as possible) and then matches $\mathcal{S}_2$ to the rest of the program. It produces a sequential composition of the programs produced by skeletons $\mathcal{S}_1$ and $\mathcal{S}_2$.

Skeletons are partial functions and a change in the abstract program could potentially make them inapplicable. In this sense, they are fragile. However, they save the programmer the work of copying code and offer a very effective mechanism of documenting critical design decisions in code. Even though our matching mechanism is deterministic, Chalice also lets the programmer inspect the final concrete code after applying all the skeletons.

Skeletons are by no means the only way to communicate structural similarity between concrete and abstract code to our verification algorithm. One could imagine using statement labels to explicitly map statements or basic support from an integrated development environment (IDE) that would permit writing refinement blocks visually as nested code blocks.

5 Related Work

As we mentioned in the introduction, refinement has a rich literature and can be described in a beautiful lattice-theoretic framework [8]. The idea of reasoning about data structures abstractly and hiding their concrete manifestations was used extensively in, for example, SIMULA [18] and CLU [35]. Hoare [26] suggested the use of a coupling invariant (aka representation invariant) to describe the connection between the abstract and concrete views. Hoare's treatment and, as far as we know, all subsequent treatment

of data refinement (*e.g.*, [27, 39]) do not consider refinements into new objects of previously defined classes. For example, Mikhajlova *et al.* [38, 6] consider data refinement in an object-oriented language, but their coupling invariants only relate the fields in a class and a subclass, not any other objects in the heap accessible via those fields.

Several tools are available for refinement. The Rodin tool set [4] includes an impressive assortment of development and testing facilities. At its core is the Event-B formalism [3], which in turn draws from action systems [7]. The executable part of an Event-B program consists of a set of guarded multi-assignment statements. This makes refinement checking much simpler than if the events had a more complex structure. Designed to handle concurrency, sequential control flow has to be encoded manually by introducing state variables. In contrast, our language uses common programming constructs like sequential composition, if and while statements, and method calls. While pointers and fields can be encoded in Event-B (*e.g.*, [1]), it does not facilitate refinements that introduce new objects of previously defined classes. Rodin provides a slick IDE in Eclipse. Its proof are mostly automatic, but frequently require some manual interaction with the proof assistant.

Atelier B [16] is a refinement tool set that supports both the Event-B and B formalisms [0]. In B, programs are sequential and hierarchically structured, like in Ada. Indeed, once programs have been refined into sufficient detail, the system can produce executable Ada or C code. Atelier B and its support tools have been put to impressive use [2]. As in Rodin, it does not facilitate refinements that introduce new objects of previously defined classes, and conducting proofs requires manual interaction with the proof assistant.

Perfect Developer is a refinement-based language and IDE for developing object-oriented programs [22]. Its strength lies in inlining objects (*i.e.*, treating classes as records), where the well-studied rules for data refinement apply. One can also use a mode where objects are instead accessed via pointers (as usual in object-oriented programs), but then its custom-built prover, which is automatic and does not permit manual intervention, can easily get stuck [14]. In this mode, the support or soundness of refinement into new objects is not clear to us.

While research on refinement has not focused on how object references are introduced and used, a lot of verification research, especially in the last decade, has. The central problem occurs when two objects are *abstractly aliased* [33], meaning that one is used as part of the internal representation of the other. In such cases, a modification of one object can affect the other, and a verification system must be able to detect or prevent such possibilities.

For this purpose, there are specification and verification techniques like ownership (*e.g.*, [15]), dynamic frames [28], separation logic [41], and implicit dynamic frames [42]; for a comparison of these techniques, see [25]. The condition that describes the consistent states of an object's data representation is called an *object invariant*. In verification, it becomes necessary to keep track of whether or not object invariants hold, which, due to the possibility of reentrancy, is not necessarily just the boundaries of public methods (see [11]). The *frame* of a method describes which parts of the program state the method may modify. In verification, it is also necessary to know the frames of methods, because the frame of an object is not necessarily entirely hid-

den from clients. Object invariants and framing complicate the specifications one has to write to do verification.

For refinement, there is hope that these specifications can be made simpler. The reason is that in the abstract view of a program, the representation of an object is not yet conceived, and therefore there is no abstract aliasing, object invariants do not relate the fields of multiple objects, and frames are just subsets of the abstract variables.

In recent work, Tafat *et al.* [43] consider data refinement in an object-oriented language. Building on a specification methodology that uses ownership, they treat the abstract state as *model fields* [30] and propose a syntax for specifying abstract witnesses when a non-deterministic coupling invariant is used. They limit refinements to one step, between an abstract level given as a pre- and postcondition specification and a concrete level given as code. The up-side of this limitation is that it makes it easier to generate first-order verification conditions, since a formula like $P \Rightarrow \mathsf{wp}[\![S, Q]\!]$ can be used. In their setting, it is necessary to include preconditions that say whether or not object invariants hold, so the hope that specifications may become simpler than for verification is not fully realized. They do not provide an extensive treatment of framing.

6 Conclusions

We have presented a refinement system that allows objects to be refined into aggregate objects and whose reasoning engine is built on a powerful SMT solver. Additionally, the language uses features common in object-oriented languages, coupling invariants can mention multiple objects, it is possible to supply abstract witnesses for non-deterministic coupling invariants, and refinement steps can be prescribed using a duplication-saving syntax of code skeletons.

We have implemented a prototype checker by incorporating the refinement features in Chalice. In the future, we would like to gain more experience with this prototype.

Our work also suggests some other research to be done. It would be interesting to explore the possibility of including language features like instantiable classes in a well-developed refinement tool like Rodin. Given the analogous features of Chalice and separation logic, our refinement framework could perhaps be adapted for use in separation logic. The language and specifications in Chalice were designed to support concurrency, so we imagine that it would be interesting to combined those features with refinement. Finally, we expressed a hope that refinement specifications could work out to be simpler than the specifications one needs for more traditional verification; we would love to see that issue resolved in the future.

Acknowledgments We are grateful to Peter Müller who suggested we might try to base our refinements on the permissions in Chalice rather than on the dynamic frames of Dafny [29], where we had started.

References

0. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

1. Jean-Raymond Abrial. Event based sequential program development: Application to constructing a pointer program. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 51–74. Springer, September 2003.
2. Jean-Raymond Abrial. Formal methods in industry: achievements, problems, future. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *28th International Conference on Software Engineering (ICSE 2006)*, pages 761–768. ACM, May 2006.
3. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
4. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, April 2010.
5. R.-J. R. Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, University of Helsinki, 1978. Report A-1978-4.
6. Ralph-Johan Back, Anna Mikhaljova, and Joakim von Wright. Class refinement as semantics of correct object substitutability. *Formal Aspects of Computing*, 12(1):18–40, October 2000.
7. Ralph-Johan Back and Kaisa Sere. Stepwise refinement of action systems. *Structured Programming*, 12(1):17–30, 1991.
8. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.
9. Thomas Ball, Brian Hackett, Shuvendu K. Lahiri, Shaz Qadeer, and Julien Vanegue. Towards scalable modular checking of user-defined properties. In Gary T. Leavens, Peter O'Hearn, and Sriram K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments, (VSTTE 2010)*, volume 6217 of *Lecture Notes in Computer Science*, pages 1–24. Springer, August 2010.
10. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.
11. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
12. Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. `http://frama-c.com/`.
13. John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.
14. Gareth Carter, Rosemary Monahan, and Joseph M. Morris. Software refinement with Perfect Developer. In Bernhard K. Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, pages 363–373. IEEE Computer Society, September 2005.
15. Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002*, pages 292–310. ACM, November 2002.
16. ClearSy. Atelier B. `http://www.atelierb.eu/`.
17. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius

Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, August 2009.

18. Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. Common base language. Publication S-22, Norwegian Computing Center, October 1970.
19. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, March–April 2008.
20. E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8:174–186, 1968.
21. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
22. Escher Technologies, Inc. Getting started with Perfect. `http://www.eschertech.com`, 2001.
23. David Gries and Jan Prins. A new notion of encapsulation. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, number 7 in SIGPLAN Notices 20, pages 131–139. ACM, July 1985.
24. David Gries and Dennis Volpano. The transform — a new language construct. *Structured Programming*, 11(1):1–10, 1990.
25. John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. Technical Report CS-TR-09-011, University of Central Florida, School of EECS, 2009.
26. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.
27. Cliff B. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.
28. Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, August 2006.
29. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, April 2010.
30. K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130. Springer, March 2006.
31. K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393. Springer, March 2009.
32. K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In Alessandro Aldini, Gilles Barthe, and Robert Gorrieri, editors, *Foundations of Security Analysis and Design V: FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer, 2009.
33. K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.
34. K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327. Springer, March 2010.

35. Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.
36. Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul I. Pénzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *17th Conference on Advanced Research in VLSI ARVLSI '97*, pages 164–181. IEEE Computer Society, September 1997.
37. Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, 1988.
38. Anna Mikhaljova and Emil Sekerinski. Class refinement and interface refinement in object-oriented programs. In John S. Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods, 4th International Symposium of Formal Methods Europe*, volume 1313 of *Lecture Notes in Computer Science*, pages 82–101. Springer, September 1997.
39. Carroll Morgan. *Programming from Specifications*. Series in Computer Science. Prentice-Hall International, 1990.
40. Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.
41. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, July 2002.
42. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference*, volume 5653 of *Lecture Notes in Computer Science*, pages 148–172. Springer, July 2009.
43. Asma Tafat, Sylvain Boulmé, and Claude Marché. A refinement methodology for object-oriented programs. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software, Papers Presented at the International Conference*, pages 143–159, June 2010.
44. N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14:221–227, 1971.