Stepwise refinement of heap-manipulating code in Chalice

Citation: Leino, K. Rustan M., and Kuat Yessenov. "Stepwise Refinement of Heap-Manipulating Code in Chalice." Formal Aspects of Computing 24.4–6 (2012): 519–535.

As Published: http://dx.doi.org/10.1007/s00165-012-0254-3

Publisher: Springer-Verlag

Persistent URL: http://hdl.handle.net/1721.1/105892

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

DOI 10.1007/s00165-012-0254-3

Formal Aspects of Computing (2012) 24: 519–535

Formal Aspects of Computing

Stepwise refinement of heap-manipulating code in Chalice

K. Rustan M. Leino[1] and Kuat Yessenov[2]

[1] Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
[2] MIT Computer Science and Artificial Intelligence Lab, Cambridge, MA, USA

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 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 imperative features, including sequential composition, loops, and recursive calls, offers a syntax with skeletons for describing program changes between refinements, and provides a mechanism for supplying witnesses when refining non-deterministic programs.

Keywords: Stepwise refinement; Data refinement; Heap refinement; Chalice; Abstract predicates; Fractional permissions; Program verification

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 white-boards 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 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.

Correspondence and offprint requests to: K. R. M. Leino, E-mail: leino@microsoft.com

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 giving various levels of pseudo-code until the low-level code is in place. This is an old idea due to Dijkstra [Dij68] and Wirth [Wir71] and given mathematical rigor by Back [Bac78]. It underwent much theoretical development during the 1980s and 1990s (e.g., [BvW98, Mor87, GV90, Jon90, WD96, Abr96]), prominently including Carroll Morgan's work on programming from specifications [Mor90]. The technique has been successfully applied in practice where program correctness has been critical (e.g., [Abr06, Cle, MLM+97]). Some tool sets, like Rodin [ABH+10] and Atelier B [Cle], 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 engineers 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 a class-based 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 aware of only one previous treatment of refinement that allows abstract fields to be refined in a general way into new objects of instantiable classes [FOTSY10].

On the verification side, we integrate automatic verification support, like that found in leading-edge program verifiers (e.g., [CDH+09, BFM+09, BHL+10, BFL+11, Lei10]), 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 view of heap-manipulating code, based on a model of memory permissions, that 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 [LM09, LMS09], which uses the Boogie verification engine [LR10] and the Z3 SMT solver [dMB08].

Our prototype implementation explores the vision of refinement where the programmer interacts with the tool solely via the constructs of the programming language, but it is not the final word. We hope that it will continue to inspire improvements in the program development process to make programs easier to write, maintain, and reason about.

In Sect. 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 Sect. 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 syntactic *skeletons* facility in Sect. 3 and our checking algorithm in Sect. 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. In other words, 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, and in this paper we ignore issues of termination. We also ignore other important properties of programs, such as information flow [DD77, Mor12]. 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 as a whole.

```
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 a given sequence. The method `Find` has an in-parameter `s` and an out-parameter `b`

In Sect. 1.1, 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.[1] In Sect. 1.2, we describe a problem with data refinement and objects.

1.1. Algorithmic refinement

1.1.1. 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 postcondition specification à la Eiffel [Mey88], 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 := e]` (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.[2]

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.[3] 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.

1.1.2. 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.

[1] http://research.microsoft.com/verificationcorner.

[2] Other starting points are also possible; for example, the assignment statement `b := exists i,j in [0..|s|] :: i != j ∧ s[i] == s[j];`.

[3] Omissions and other errors in the top-level specification may become evident when clients of the class are verified (using Chalice). Moreover, various techniques and tools exist for checking that a software specification or model has desired properties (among many, see, e.g., [Jac06, LB03]). Such checks could be applied to the top-level specification in the refinement, but we do not focus on such checks here.

```
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

```
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. The occurrences of "_", and also the **`while`** statement without a loop guard, are concrete syntax in our language and stand for the corresponding pieces of code in the method being transformed

The transformation from Figs. 0 to 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 Figs. 0 or 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.

1.1.3. 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.

Suppose the specification of the original program is revisited and we are now provided with a restriction on the input, namely that the elements of `s` are to be among the first 100 natural numbers. We express the restriction by the following precondition, which we add to the program in Fig. 0:

```
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 [Abr06].

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`** swe 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 Sect. 3. 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* [Mor90]). 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.

1.1.4. Summarizing the example

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

1.2. 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 occurs in the presence of pointers and instantiable object libraries [FOTSY10].

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.

In Fig. 4, we show a client of the `Counter` class. It allocates a `Counter` object. 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.[4] 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()`.

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

```
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

```
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 evaluate to `true`

In Fig. 5, we show a refinement of class `Counter`. It superimposes a field `c`, sets `c` to a new (dynamically allocated) `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., [Hoa72, Mor90, GV90]), but—surprisingly—not much in the 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 which `Cell` is returned by `M()`, giving `CCounter` total freedom in what it returns.

The problem here is that `CCounter` fails to be a valid refinement since it cannot be substituted in place of `Counter` in `Client`. We propose a solution to this problem that relies on the permission model of Chalice. The solution requires more specifications for the `Counter` methods. Depending on which specification is chosen for method `M()`, our tool will either blame the client or the refinement.

```
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 [Boy03] and implicit dynamic frames [SJP09]. This model forms a core of the language and verifier Chalice [LM09, LMS09], into which we have incorporated our refinement system. Chalice and our extensions are available as open source[5] and can be run either from the command line or from within the Microsoft Visual Studio IDE.

2.1. 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 the method is written 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);`**.

Specifications can mention several *access predicates*, which are evaluated in order. For example, suppose a method declares the precondition **`acc(x)`** $\wedge$ **`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.

We say that the caller *exhales* the precondition, meaning that it checks the conditions and access predicates in the precondition and transfers the entailed permissions to the callee. The callee *inhales* the precondition, meaning that it gets to assume the conditions in the precondition and receives the entailed permissions.

Note that a specification like **`acc(p.x)`** $\wedge$ **`acc(q.x)`** is satisfiable only if `p` and `q` are pointers to different objects. For if `p` and `q` are equal, then `p.x` and `q.x` denote the same heap location. Thus, after the permission to `p.x` has been (checked and) transferred, then the check for permission to `q.x` will fail.

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`. One can also simply write **`rd(x)`** to denote a non-zero permission to `x` and leave it to the verifier to infer an appropriate fraction; we will not describe the details here, but see [HLMS11].

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)`** $\wedge$ **`acc(x,50)`** is equivalent to **`acc(x)`**, since the two fractions add up to the full permission; and the condition **`acc(x,80)`** $\wedge$ **`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 record 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 [LM09, HLMS11].

Consider the `Counter` example in Sect. 1.2. 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. In particular, it does not say or imply anything about the permission to this `Cell`'s `x` field, and therefore the verifier will report an error that the client code attempts to modify a heap location (namely, `cell.x`) to which it has no permissions.

[5] http://boogie.codeplex.com

Alternatively, if we want callers of `M()` to be able to (read or) modify the `x` field of the `Cell` returned, we can change `M()` accordingly:

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

This postcondition gives the caller full permission to `r.x`, and thus `Client` verifies. Note that we also updated the specification statement in the body of `M()`, which now says to pick not just any `Cell` for `r`, but one for which full permission of `x` is available, and thus `Counter` verifies.

2.2. Coupling invariants for the heap refinement

In our running example, the field `n` of `Counter` is represented with `c.x` in the refinement. The relationship between the *abstract location* `n` and the *concrete location* `c.x` is captured in the coupling invariant declaration inside the refinement class `CCounter`:

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

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

If only a fractional permission to `n` is traded, then the permissions to `c` and `c.x` are scaled accordingly. This is done by multiplying the permissions mentioned in the coupling invariant by the fraction of `n`'s permission that is being traded.

Let's go back to the `CCounter` example. As written in Fig. 5, the refinement of `M` fails to verify since assignment `r := c` has insufficient permissions to read `c`. Now imagine that we add the precondition **acc**`(n)` to `M` in class `Counter` (we cannot add **acc**`(c)` directly, since `c` is declared in the refinement). We should also add the postcondition **acc**`(n)` or, otherwise, the client is not able to inspect `n` after making a call to `M`:

```
method M() returns (r: Cell)
requires acc(n);
ensures acc(n) ∧ acc(r.x);
```

However, even with these permissions in place, the refinement `M` fails to verify since both **acc**`(n)` and **acc**`(r.x)` individually imply full access to `c.x`. Since the postcondition is never satisfiable, our tool reports that `CCounter` does not refine `Counter`.

3. Surface syntax

In this section, we present our extensions to the syntax of Chalice [LMS09] to support program refinement. These extensions include class and method refinement declarations, coupling invariants, and program structure skeletons.

3.1. Class refinement

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

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

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 in the context of this refinement (A may in turn be a refinement as well). 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. B may add methods which are not present in A and may superimpose fields.

```
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) A program that computes $\sum_{i=0}^{n} i^3$

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

(b) A skeleton that transforms the original program in Fig. 6a into the program in Fig. 6c

```
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;
  }
}
```

(c) A program that establishes the identity $\sum_{i=0}^{n} i^3 = \left(\sum_{i=0}^{n} i\right)^2$ as a loop invariant

Fig. 6. Refinement of a program that computes the sum of cubes. The *highlighted lines* show the new code in the program in 7c

1. B may declare a class-wide *coupling invariant* I to simultaneously replace fields f_i of A:

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

   The access predicates inside I are split evenly between f_i. The coupling invariant grants an activation record with write access to a field f_i, a k-th fraction of permissions to the concrete representation.
2. B may declare a method m to be a refinement of a method m in A using either the **refines** or **transforms** keyword instead of the **method** keyword.

It is often the case that 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. The example in Fig. 6 demonstrates one such scenario: the insight behind this refinement is the mathematical identity $\sum_{i=0}^{n} i^3 = \left(\sum_{i=0}^{n} i\right)^2$ that lets one compute a sum of cubes with a single multiplication. To verify this optimization, a programmer needs to introduce a new local variable `t` and establish coupling with the variable `s` using a loop invariant. This transformation is succinctly expressed in Chalice as a *skeleton*, see Fig. 6b.

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

3.2. Skeletons

Skeletons are transformation rules that are composed of code navigation and rewrite operations. Given an abstract program, a 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, a skeleton is a partial function from an abstract syntax tree (AST) of an abstract program to an AST of the concrete program. Since a skeleton maintains the original control flow structure, it helps us to think of each statement of the resulting program as being one of:

- a normal statement
- a *refinement block* $R[A, B]$, which replaces a sequence A of abstract statements by a sequence B of concrete statements
- a *loop refinement* $L[I, Q]$, which adds a loop invariant I to an existing loop and replaces its body by Q

In a well-formed refinement block $R[A, B]$, B declares all the local variables declared in A. Our checking algorithm benefits from the localized verifications arising from 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:

- A skeleton _ is a block pattern that matches any sequence of non-conditional non-iterative deterministic statements and acts as an identity transformation.
- A skeleton $*$ matches any sequence of statements and acts as an identity transformation.
- A skeleton **replaces** $*$ **by** { B } matches any sequence of statements A and produces $R[A, B]$.
- A skeleton **if** { $\mathcal{S}_0$ } matches a single **if** statement and produces an **if** statement with $\mathcal{S}_0$ applied to its branch; a skeleton **if** { $\mathcal{S}_0$ } **else** { $\mathcal{S}_1$ } is analogous.
- A skeleton **while invariant** I { $\mathcal{S}_0$ } 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 a skeleton $\mathcal{S}_0$ to the body of the original loop. We use the notation $L[I, P]$ for such a loop refinement.
- A skeleton **replaces** v **by** { B } matches any statement A that affects variables in list v. The resulting refinement block is $R[A, B]$. This pattern is used to provide witnesses to nondeterministic specifications or call statements and to rewrite assignment statements. Our checking algorithm resolves angelic non-determinism as described in Sect. 4.
- A skeleton B consisting of Chalice statements matches only the empty program and produces B.
- A sequential skeleton $\mathcal{S}_0;\mathcal{S}_1$ matches $\mathcal{S}_0$ greedily (i.e. consuming as many statements as possible) and then matches $\mathcal{S}_1$ against the rest of the program. It produces a sequential composition of the results of $\mathcal{S}_0$ and $\mathcal{S}_1$.

Note that skeletons with wildcards match greedily. This makes the matching process deterministic and avoids the need for back tracking.

Skeletons are partial functions, so that 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 an 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 by a command-line switch.

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.

4. Checking algorithm

Our system leverages 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., [dMB08]), to reason about program refinements. How to produce input for such a reasoning engine is well known (see, e.g., [BCD+06]): 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 [Dij76]. 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 the weakest preconditions as

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

where the quantification of Q ranges over all predicates [Bac78]. 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.

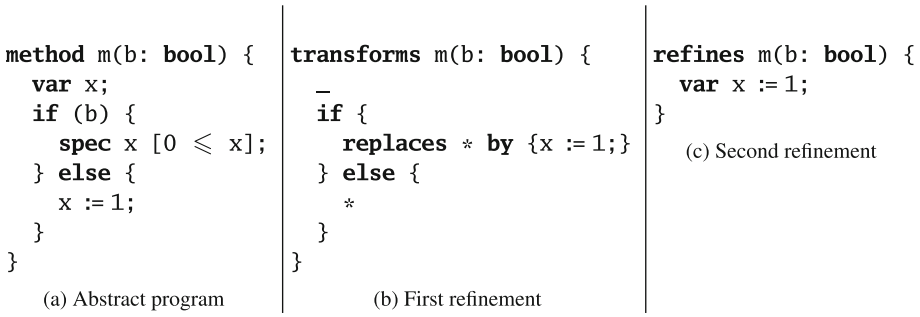

Fig. 7. Refinement of a non-deterministic program in two steps. To establish refinement of the program in **a** by the program in **c**, Chalice requires the intermediate program in **b**, which solely refines the specification statement

The refinement condition (3) is expressible in a different form that avoids predicate quantification using a *coupling invariant* I [GP85]:

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

The condition $\mathsf{wp}[\![A, \neg I]\!]$ characterizes the set of states from which A is guaranteed to reach $\neg I$, and hence $\neg\mathsf{wp}[\![A, \neg I]\!]$ is the set of states from which there is some possible execution of A that does not establish $\neg I$. In other words, formula (4) says that 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. This is also known as a *forward simulation* [WD96].

Here is an alternative explanation of formula (4), thinking of programs and the coupling invariant as being relations on pairs of states. Let σ and σ' range over abstract states (that is, A's state space) and let τ and τ' range over concrete states (that is, B's state space). Then, formula (4) says:

$$\begin{array}{l}(\,\forall \sigma, \tau \bullet\ (\sigma, \_) \in A \wedge (\sigma, \tau) \in I \;\Rightarrow \\ \qquad (\tau, \_) \in B \wedge (\,\forall\, \tau' \bullet\ (\tau, \tau') \in B \;\Rightarrow\; (\,\exists \sigma' \bullet\ (\sigma, \sigma') \in A \wedge (\sigma', \tau') \in I\ )))\end{array}$$

where we have used the notation $(s, \_) \in S$ as a shorthand for $(\,\exists s' \bullet\ (s, s') \in S\,)$.

If A is deterministic, then any execution is angelic and we can cancel the double negation in formula (4), and simplify it to:

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

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 is declared for the superimposed heap locations using the **replaces** keyword. The local variables of A are bound to the local variables of B via logical equality. The superimposed local variables in program B are left unconstrained by I.

If the program A is non-deterministic, then formula (5) is a sound but not complete characterization of refinement. Chalice provides two ways to introduce (demonic) 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 amounts to extracting witnesses from B that satisfy the postcondition of A. To provide such witnesses, the program B must assign to the abstract variables that are constrained by the postcondition of A within the refinement block, in which case it suffices to check that the assigned values (which might be demonically non-deterministic) satisfy this postcondition at the end of the block. We have imposed a restriction that non-deterministic statements must be refined individually since Chalice uses automated first-order SMT solvers. A consequence of this restriction is that refinement of non-deterministic constructions might require multiple steps, as demonstrated in figure Fig. 7.

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

0. `B.m` has the same precondition but possibly a stronger postcondition.
1. `B.m` accepts the same number of 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 Sect. 3) allows us to compute the correspondence between abstract statements of `A.m` and concrete statements of `B.m`. Once code substitutions are localized to the disjoint refinement blocks $R[P, Q]$ and loop refinements $L[I, Q]$, Chalice generates a Boogie program C [BCD+06, LR10] that encodes the refinement condition. 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`. Program C takes the same inputs as `A.m` and produces the same outputs as `B.m`. The superimposed fields of class `B` and new local variables in `B.m` are declared in C. Translation then proceeds by inserting refinement checks into the refinement blocks.

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 the Boogie intermediate language. These steps bear resemblance to formula (5, but with exhale and inhale in place of assert and assume, and a way to allow witnesses to resolve non-determinism.

0. Duplicate the state (the heap, the permission mask, and the local variable environment).
1. Permissions to access the superimposed fields are derived from the permissions of the replaced fields by splitting the fractional access permissions inside the invariant evenly among the replaced fields (see the general rule for the replaced fields in Sect. 3).
   The coupling invariant is then scaled by the amount of permissions to the replaced field held in the primary copy times its share of the invariant, and *inhaled* into the secondary copy. The inhale operation in Chalice transfers permissions to the callee and assumes the truth of the logical condition similar to assume-guarantee reasoning in the sequential verification [LM09, HLMS11]. Permissions to access the replaced fields are analogously *exhaled* from the secondary copy.
2. Execute Q from `B.m` normally using the secondary copy of the state.
3. Execute P from `A.m` angelically using the primary copy of the state. Here Chalice faces a limitation as it is only capable of expressing in Boogie angelic execution of deterministic programs, call statements, and specification statements. If P is a single non-deterministic statement, then Chalice replaces P with

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

   where $post[P]$ is the postcondition of P. This essentially lets Q compute a witness that is then checked to satisfy $post[P]$.
4. Check the coupling between the two copies of the state. For the local variables from `A.m`, assert the logical equality. The coupling invariant is scaled again by the amount of permissions to the replaced field held at the end of P times its share, and *exhaled* from the secondary copy. The exhale operation checks that the copy holds sufficient amount of permissions and asserts the truth of the logical condition.
   If there are multiple refinement blocks in `B.m`, the values of the superimposed fields and local variables are carried over to the subsequent refinement blocks.

Loop refinement $L[I, Q]$ Chalice adds assertions to establish the new loop invariant I at the entrance of the loop and to show that the body Q maintains it. The body of the loop itself might contain refinement blocks and loop refinements.

Figure 8 shows a simplified encoding into Boogie of the refinement of the program for computing the sum of cubes from Fig. 6. The two additional loop invariants are assumed at the beginning of the loop body and asserted at the end. The Boogie program is fed through the Boogie tool's pipeline and into the automated theorem prover Z3 [dMB08].

Modular refinement Chalice can optionally assume correctness of the abstract program to prove refinement of the concrete program. All pre-existing assertions of A are eliminated from C by turning them into assumptions, provided the updates to those variables have not changed. For example, the loop invariant $\texttt{i} \leqslant \texttt{n}$ from the abstract version of the program in Fig. 6 is assumed throughout the encoding in Fig. 8.

The technique effectively modularizes the verification of the program into refinement steps. We hope that by structuring specifications and code into refinements, we can also better tackle the verification of programs that without the refinement extension are challenging to Chalice.

```
  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
    }
  }
}
```

Fig. 8. Refinement condition in Boogie of a program for computing the sum of cubes from Fig. 6

5. Related work

Refinement has a rich literature, see for example the references we mentioned in the introduction, and can be described in a beautiful lattice-theoretic framework [BvW98] (and see also [Mor94]). The idea of reasoning about data structures abstractly and hiding their concrete manifestations was used extensively in, for example, SIMULA [DMN70] and CLU [LG86]. Hoare [Hoa72] suggested the use of a coupling invariant (aka representation invariant) to describe the connection between the abstract and concrete views.

Hoare's treatment and most subsequent treatments of data refinement (e.g., [Jon90, Mor90]) do not consider refinements into new objects of previously defined classes. For example, Mikhajlova et al.[MS97, BMvW00] 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. Similarly, Grandy et al. use the KIV tool to do refinement for Java programs, but the concrete programs contain all new classes, not instances of previously defined classes [GSR07].

As far as we know, only Filipović et al. have spelled out and tackled the general problem of data refinements into dynamic storage before [FOTSY10]. We have reached the same conclusions as they, that a client is not allowed to update a heap location just because it is able to reach that heap location. That is, just because a client is able to compute an address in the heap (even if the programming language can guarantee something about the type of the value stored in that heap location, which is the case for `cell.x` in Fig. 4) does not mean the client is allowed to access that heap location. When no pointers are involved, it is reasonable to restrict the concrete representation used in a data refinement to variables or fields introduced as part of the refinement. But to let the

concrete representation include instances of previously declared classes, it is necessary to generalize the notion of "variables used by the concrete representation" to "heap locations used by the concrete representation". Both Filipović et al. and we achieve this.

The biggest difference between the work of Filipović et al. and ours lies in how the coupling relation is provided. In particular, we make the coupling invariant part of the concrete program text, and we make it explicit which abstract variables are being replaced. Also, our work includes an implementation in a tool.

Our coupling invariants (like (1)) declare a relation on not just the values of abstract and concrete variables, but also on the access permissions to these variables. This is quite related to the treatment of abstract predicates [PB05], as implemented for example in VeriFast [JP08] and Chalice [LMS09, HKMS12]. In particular, entering the concrete view corresponds to *opening* (aka *unfolding*) a predicate, and returning to the abstract view corresponds to *closing* (aka *folding*) the predicate. Two differences are that our abstract variables need not have boolean type (so they are more like *model fields*, see, e.g., [HLL+12]) and our coupling-invariant declarations allow a list of abstract variables to be replaced by concrete ones.

Jones's work on $\pi o \beta \lambda$ includes data refinement and does allow the concrete program to allocate and make use of new objects of previously defined classes [Jon96]. However, this is achieved by the draconian rule that an object reference is not allowed to be passed "over" another, meaning that methods cannot take object references as in- or out-parameters.

Several tools are available for refinement. The Rodin tool set [ABH+10] includes an impressive assortment of development and testing facilities. At its core is the Event-B formalism [Abr10a], which in turn draws from action systems [BS91]. 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. Although the Chalice language and verifier support concurrency, we have not investigated the connection between these features and refinement, whereas concurrency and refinement have always been present in Event-B. While pointers and fields can be encoded in Event-B (e.g., [Abr03]), it does not facilitate refinements that introduce new objects of previously defined classes. Rodin provides a slick IDE in Eclipse. Its proof assistant is mostly automatic, but frequently requires some manual interaction with the proof assistant.

Atelier B [Cle] is a refinement tool set that supports both the Event-B and B formalisms [Abr96]. 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 [Abr06]. 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 [Esc01]. 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 [CMM05]. In this mode, the support for and 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* [LN02], 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., [CD02]), dynamic frames [Kas06], separation logic [Rey02], and implicit dynamic frames [SJP09]; for a comparison of these techniques, see [HLL+12]. The condition that describes the consistent states of an object's data representation is called a *class invariant* [Mey88]. In verification, it becomes necessary to keep track of whether or not a class invariant holds, which, due to the possibility of reentrancy, is not necessarily just the boundaries of public methods [BDF+04]. 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 hidden from clients. Class 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, class invariants do not relate the fields of multiple objects, and frames are just subsets of the abstract variables.

A common discipline in object-oriented programming is that subclasses are to be used as behavioral subtypes. This discipline has been formalized, with some variations in the treatment of one- and two-state class invariants [Lea91, LW94, SH02]. Behavioral subtyping is closely related to refinement in that it describes when one class can be replaced by a subclass.

In recent work, Tafat et al. [TBM10] consider data refinement in an object-oriented language. Building on a specification methodology that uses ownership, they treat the abstract state as *model fields* [LM06] 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 class 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.

The impressive verification of the seL4 operating system kernel also makes use of data refinement [KSW10]. It seems that their refinements could be extended to our way of dealing with refinements into new instances of previously existing classes, because the seL4 model includes memory capabilities, which are like the permissions used by Chalice.

Finally, we mention that Event-B also has a syntactic construct for specifying a witness when an event is refined [Abr10b].

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. 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. So far, we have applied the prototype only to smaller examples, partly due to the fact that Chalice currently does not support sets or maps, which often occur in refinement examples. 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. The language and specifications in Chalice were designed to support concurrency, so we imagine that it would be interesting to combine 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

This work was performed while Kuat Yessenov was doing a research internship at Microsoft Research. 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 [Lei10], where we had started. We thank Emil Sekerinski and the referees for comments on drafts of this paper.

References

[ABH+10] Abrial J-R, Butler M, Hallerstede S, Hoang TS, Mehta F, Voisin L (2010) Rodin: an open toolset for modelling and reasoning in Event-B. Int J Softw Tools Technol Transf

[Abr96] Abrial J-R (1996) The B-Book: assigning programs to meanings. Cambridge University Press, Cambridge

[Abr03] Abrial J-R (2003) Event based sequential program development: Application to constructing a pointer program. In: Araki K, Gnesi S, Mandrioli D (eds) FME 2003: formal methods, international symposium of formal methods Europe. Lecture Notes in Computer Science, vol 2805. Springer, Berlin, pp 51–74

[Abr06] Abrial J-R (2006) Formal methods in industry: achievements, problems, future. In: Osterweil LJ, Dieter Rombach H, Soffa ML (eds) 28th international conference on software engineering (ICSE 2006). ACM, New York, pp 761–768

[Abr10a] Abrial J-R (2010a) Modeling in Event-B: system and software engineering. Cambridge University Press, Cambridge

[Abr10b] Abrial J-R (2010b) Modeling in Event-B: system and software engineering. Cambridge University Press, Cambridge

[Bac78] Back RJR (1978) On the correctness of refinement steps in program development. PhD thesis, University of Helsinki. Report A-1978-4.
[BCD+06] Barnett M, Chang B-YE, DeLine R, Jacobs B, Leino KRM (2006) Boogie: a modular reusable verifier for object-oriented programs. In: de Boer FS, Bonsangue MM, Graf S, de Roever W-P (eds) Formal methods for components and objects: 4th international symposium, FMCO 2005. Lecture Notes in Computer Science, vol. 4111. Springer, Berlin, pp 364–387
[BDF+04] Barnett M, DeLine R, Fähndrich M, Leino KRM, Schulte W (2004) Verification of object-oriented programs with invariants. J Object Technol, 3(6):27–56
[BFL+11] Barnett M, Fähndrich M, Leino KRM, Müller P, Schulte W, Venter H (2011) Specification and verification: the Spec# experience. Commun. ACM, 54(6):81–91
[BFM+09] Baudin P, Filliâtre JC, Marché C, Monate B, Moy Y, Prevosto V (2009) ACSL: ANSI/ISO C specification language, version 1.4. http://frama-c.com/
[BHL+10] Ball T, Hackett B, Lahiri SK, Qadeer S, Vanegue J (2010) Towards scalable modular checking of user-defined properties. In: Leavens GT, O'Hearn P, Rajamani SK (eds) Verified software: theories, tools, experiments, (VSTTE 2010). Lecture Notes in Computer Science, vol 6217. Springer, Berlin, pp 1–24
[BMvW00] Back R-J, Mikhaljova A, von Wright J (2000) Class refinement as semantics of correct object substitutability. Formal Aspects Comput 12(1):18–40
[Boy03] Boyland J (2003) Checking interference with fractional permissions. In: Cousot R (ed) Static analysis, 10th international symposium, SAS 2003. Lecture Notes in Computer Science, vol 2694. Springer, Berlin, pp 55–72
[BS91] Back R-J, Sere K (1991) Stepwise refinement of action systems. Struct Program 12(1):17–30
[BvW98] Back R-J, von Wright J (1998) Refinement calculus: a systematic introduction. Graduate Texts in Computer Science. Springer, Berlin
[CD02] Clarke D, Drossopoulou S (2002) 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. ACM, New York, pp 292–310
[CDH+09] Cohen E, Dahlweid M, Hillebrand M, Leinenbach D, Moskal M, Santen T, Schulte W, Tobies S (2009) VCC: a practical system for verifying concurrent C. In: Berghofer S, Nipkow T, Urban C, Wenzel M (eds) Theorem proving in higher order logics, 22nd international conference, TPHOLs 2009. Lecture Notes in Computer Science, vol 5674. Springer, Berlin, pp 23–42
[Cle] ClearSy. Atelier B. http://www.atelierb.eu/.
[CMM05] Carter G, Monahan R, Morris JM (2005) Software refinement with perfect developer. In: Aichernig BK, Beckert B (eds) Third IEEE international conference on software engineering and formal methods (SEFM 2005). IEEE Computer Society, New York, pp 363–373
[DD77] Denning DE, Denning PJ (1977) Certification of programs for secure information flow. Commun ACM 20(7):504–513
[Dij68] Dijkstra EW (1968) A constructive approach to the problem of program correctness. BIT 8:174–186
[Dij76] Dijkstra EW (1976) A discipline of programming. Prentice Hall, Englewood Cliffs
[dMB08] de Moura L, Bjørner N (2008) Z3: an efficient SMT solver. In: TACAS 2008. Lecture Notes in Computer Science, vol 4963. Springer, Berlin, pp 337–340
[DMN70] Dahl O-J, Myhrhaug B, Nygaard K (1970) Common base language. Publication S-22, Norwegian Computing Center
[Esc01] Escher Technologies, Inc. (2001) Getting started with perfect. http://www.eschertech.com
[FOTSY10] Filipović I, O'Hearn P, Torp-Smith N, Yang H (2010) Blaming the client: on data refinement in the presence of pointers. Formal Aspects Comput 22(5):547–583
[GP85] Gries D, Prins J (1985) A new notion of encapsulation. In: Proceedings of the ACM SIGPLAN 85 symposium on language issues in programming environments. SIGPLAN Notices, vol 20, No. 7. ACM, New York, pp 131–139
[GSR07] Grandy H, Stenzel K, Reif W (2007) A refinement method for Java programs. In: Bonsangue MM, Johnsen EM (eds) Formal methods for open object-based distributed systems, 9th IFIP WG 6.1 international conference, FMOODS 2007. Lecture Notes in Computer Science, vol 4468. Springer, Berlin, pp 221–235
[GV90] Gries D, Volpano D (1990) The transform—a new language construct. Struct Program 11(1):1–10
[HKMS12] Heule S, Kassios IT, Müller P, Summers AJ (2012) Verification condition generation for permission logics with abstraction functions. Technical Report 761, ETH Zurich
[HLL+12] Hatcliff J, Leavens GT, Rustan M. Leino K, Müller P, Parkinson M (2012) Behavioral interface specification languages. ACM Comput Surv, 44(3)
[HLMS11] Heule S, Rustan M. Leino K, Müller P, Summers AJ (2011) Fractional permissions without the fractions. In: 13th workshop on formal techniques for Java-like programs, FTfJP 2011
[Hoa72] Hoare CAR (1972) Proof of correctness of data representations. Acta Informatica 1(4):271–281
[Jac06] Jackson D (2006) Software abstractions: logic, language, and analysis. MIT Press, Cambridge
[Jon90] Jones CB (1990) Systematic software development using VDM. International Series in Computer Science, 2nd edn. Prentice Hall, Englewood Cliffs
[Jon96] Jones CB (1996) Accommodating interference in the formal design of concurrent object-based programs. Formal Methods Syst Des 8(2):105–122
[JP08] Jacobs B, Piessens F (2006) The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven
[Kas06] Kassios IT (2006) Dynamic frames: support for framing, dependencies and sharing without restrictions. In: Misra J, Nipkow T, Sekerinski E (eds) FM 2006: formal methods, 14th international symposium on formal methods. Lecture Notes in Computer Science, vol 4085. Springer, Berlin, pp 268–283
[KSW10] Klein G, Sewell T, Winwood S (2010) Refinement in the formal verification of seL4. In: Hardin DS (ed) Design and verification of microprocessor systems for high-assurance applications. Springer, Berlin, pp 323–339
[LB03] Leuschel M, Butler M (2003) ProB: a model checker for B. In: Araki K, Gnesi S, Mandrioli D (eds) FME 2003: formal methods. Lecture Notes in Computer Science, vol 2805. Springer, Berlin, pp 855–874

[Lea91] Leavens GT (1991) Modular specification and verification of object-oriented programs. IEEE Softw 8(4):72–80
[Lei10] Leino KRM (2010) Dafny: an automatic program verifier for functional correctness. In: Clarke EM, Voronkov A (eds) LPAR-16. Lecture Notes in Computer Science, vol 6355. Springer, Berlin, pp 348–370
[LG86] Liskov B, Guttag J (1986) Abstraction and specification in program development. MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge
[LM06] Leino KRM, Müller P (2006) A verification methodology for model fields. In: Sestoft P (ed) Programming languages and systems, 15th European symposium on programming, ESOP 2006. Lecture Notes in Computer Science, vol 3924. Springer, Berlin, pp 115–130
[LM09] Leino KRM, Müller P (2009) A basis for verifying multi-threaded programs. In: Castagna G (ed) Programming languages and systems, 18th European Symposium on Programming, ESOP 2009. Lecture Notes in Computer Science, vol 5502. Springer, Berlin, pp 378–393
[LMS09] Leino KRM, Müller P, Smans J (2009) Verification of concurrent programs with Chalice. In: Aldini A, Barthe G, Gorrieri R (eds) Foundations of security analysis and design V: FOSAD 2007/2008/2009 tutorial lectures. Lecture Notes in Computer Science, vol 5705. Springer, Berlin, pp 195–222
[LN02] Leino KRM, Nelson G (2002) Data abstraction and information hiding. ACM Trans Program Lang Syst 24(5):491–553
[LR10] Leino KRM, Rümmer P (2010) A polymorphic intermediate verification language: design and logical encoding. In: Esparza J, Majumdar R (eds) Tools and algorithms for the construction and analysis of systems, 16th international conference, TACAS 2010. Lecture Notes in Computer Science, vol 6015. Springer, Berlin, pp 312–327
[LW94] Liskov B, Wing JM (1994) A behavioral notion of subtyping. ACM Trans Program Lang Syst 16(6)
[Mey88] Meyer B (1998) Object-oriented software construction. Series in Computer Science. Prentice-Hall, NJ
[MLM+97] Martin AJ, Lines A, Manohar R, Nyström M, Pénzes PI, Southworth R, Cummings U (1997) The design of an asynchronous MIPS R3000 microprocessor. In: 17th conference on advanced research in VLSI ARVLSI '97. IEEE Computer Society, New York, pp 164–181
[Mor87] Morris JM (1987) A theoretical basis for stepwise refinement and the programming calculus. Sci Comput Program 9(3):287–306
[Mor90] Morgan C (1990) Programming from specifications. Series in Computer Science. Prentice-Hall International, NJ
[Mor94] Morgan C (1994) The cuppest capjunctive capping, and Galois. In: Roscoe AW (ed) A classical mind: essays in honour of C.A.R. Hoare. International Series in Computer Science. Prentice-Hall, NJ, pp 317–332
[Mor12] Morgan C (2012) Compositional noninterference from first principles. Formal Aspects Comput 24(1):3–26
[MS97] Mikhaljova A, Sekerinski E (1997) Class refinement and interface refinement in object-oriented programs. In: Fitzgerald JS, Jones CB, Lucas P (eds) FME '97: industrial applications and strengthened foundations of formal methods, 4th international symposium of formal methods Europe. Lecture Notes in Computer Science, vol 1313. Springer, Berlin, pp 82–101
[PB05] Parkinson MJ, Bierman GM (2005) Separation logic and abstraction. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2005. ACM, New York, pp 247–258
[Rey02] Reynolds JC (2002) Separation logic: a logic for shared mutable data structures. In: 17th IEEE symposium on logic in computer science (LICS 2002). IEEE Computer Society, New York, pp 55–74
[SH02] Shield J, Hayes IJ (2002) Refining object-oriented invariants and dynamic constraints. In: 9th Asia–Pacific software engineering conference (APSEC 2002). IEEE Computer Society, New York, pp 52–61
[SJP09] Smans J, Jacobs B, Piessens F. Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou S (ed) ECOOP 2009—Object-oriented programming, 23rd European conference. Lecture Notes in Computer Science, vol 5653. Springer, Berlin, pp 148–172
[TBM10] Tafat A, Boulmé S, Marché C (2010) A refinement methodology for object-oriented programs. In: Beckert B, Marché C (eds) Formal verification of object-oriented software, papers presented at the international conference, pp 143–159
[WD96] Woodcock J, Davies J (1996) Using Z: Specification, refinement, and proof. Prentice Hall, NJ
[Wir71] Wirth N (1971) Program development by stepwise refinement. Commun ACM 14:221–227

Received 23 December 2011
Accepted in revised form 28 May 2012 by Peter Höfner, Robert van Glabbeek, Ian Hayes and Jim Woodcock
Published online 2 July 2012