Chapter 4

Modeling, Specification Languages, Array Programs

This chapter presents some techniques for verifying programs with complex data structures. Typical examples of such data structures are arrays and lists. The main ingredient is to provide extensions to the language of expressions and formulas, under the form of an expressive specification language.

Section 4.1 presents generalities about specification languages, while other sections present an overview of the constructs for modeling and specifying as they are available in the Why3 tool [7]. Finally, Section 4.6 is especially devoted to the modeling and the proof of programs on the array data structure.

4.1 Generalities on Specification Languages

There are a lot of variants of specification languages proposed in the past. Various classes of such languages are characterized by the kind of logic that they are based on.

Algebraic Specifications This class of languages is characterized by the use of multi-sorted first-order logic with equality, and usually the use of rewriting techniques [13], which permits to mix executable-style and declarative-style specifications. Famous languages and tools of this class are Larch [17], KIV [26], CASL (Common Algebraic Specification Language) [2], OBJ [16], Maude [10].

Set-theory based approaches Another class of languages is based on classical Set theory. These languages are not typed, the typing properties being instead encoded by formulas on sets. Reasoning on such languages is also typically done by rewriting techniques. A precursor of this class is the Z notation [27]. It is used in tools like VDM [18], Atelier B [1], etc.

Higher-Order Logic Higher-Order Logic is characterized by the use of mathematical functions as objects of the language. It is based on advanced lambda-calculi (e.g. system F). Reasoning on such specifications is typically done using interactive environment where proofs are constructed manually using tactics. Famous tools implementing some higher-order logics are PVS [24], Isabelle [22], HOL4 [23], Coq [5], etc.

Object-Oriented languages These are based on the idea of using the object-oriented paradigm in a logic world. They are typically in use in a context of object-oriented-style development. The precursor is the language Eiffel [20], which is both a programming language and a specification language, and was designed to support the concept of *design-by-contract*. Other famous languages are OCL [28] (Object

Constraint Language) which allows to incorporate formal specifications into the UML diagrammatic design methodology, and the Java Modeling Language [9].

Satisfiability Modulo Theories In 1980, Nelson and Oppen [21] proposed a new technique for combining propositional reasoning and dedicated decision procedures for specific theories. This gave birth to a new class of automated theorem provers called SMT solvers. A precursor of such solvers is Simplify [14] which was used for program verification in the ESC/Java tool. The SMT-LIB format [25] is not really a specification language but a common language for describing verification conditions as they typically come from program verification. In this context, the intermediate languages for program verification Why [15] and Boogie [3] propose small specification languages that can be supported by SMT solvers. Nowadays, the successors of Why and Boogie, respectively Why3 [7] and Dafny [19], propose modeling languages that can be seen as a compromise between:

- Expressivity of the language, to be able to design specifications of complex programs ;
- Support by automated provers, in particular modern SMT solvers like Alt-Ergo [6], Z3 [12], CVC3 [4] which support quantified first-order formulas.

The following sections present an overview of the constructs of the Why3 specification language.

4.2 Defined functions and predicates

Direct definitions of new functions and predicates can be given under the form

function $f(x_1 : \tau, \ldots, x_n : \tau_n) : \tau = e$
predicate $p(x_1 : \tau, \ldots, x_n : \tau_n) = e$

where τ_i, τ are not reference types.

In such definitions, no recursion is allowed. Since expressions have no side-effects, these define total functions and predicates, which are *pure* in the sense that they don't do side-effects.

Example 4.2.1 *Here are a few examples of definitions of functions and predicates.*

function sqr(x:int) : int = x $*$ x

function min(x:real,y:real) : real = **if** x $\leq$ y **then** x **else** y

function abs(x:real) : real = **if** x $\geq$ 0.0 **then** x **else** $-$x

predicate in_0_1(x:real) = 0.0 $\leq$ x $\wedge$ x $\leq$ 1.0

predicate prime(x:int) = x $\geq$ 2 $\wedge$ **forall** y z:int. y $\geq$ 0 $\wedge$ z $\geq$ 0 $\wedge$ x = y$*$z $\rightarrow$ y=1 $\vee$ z=1

4.3 Constructed types

New logic data types can be defined by standard construction of compound types. Being in a logic language, these types must be pure in the sense of pure functional programming, that is the values of these types cannot be modified "in-place".

4.3.1 Product Types

Types can be defined as being tuples of already defined types, for example

```
type intpair = (int, int)
function sumprod (n:int,m:int) : intpair = (n+m,n*m)
type vector = (real, real, real)
function diag(x:real) : vector = (x,x,x)
```

The construct **let** x=e1 **in** e2 is extended allowing to introduce not only a variable x but a tuple, as in

```
function sum(p:intpair) = let (x,y) = p in x+y
```

The other kind of construction of product type is the record types, defined classically by a sequence of named fields. The notation with dot allows to access a given field. For example[1]

```
type vector = { x:real; y:real }
function norm(v:vector) : real = sqrt(v.x * v.x + v.y * v.y)
function rotate(v:vector,a:real) : vector =
  { x = cos(a) * v.x + sin(a) * v.y ;
    y = sin(a) * v.y − sin(a) * v.x }
```

Again, it is important to notice that these are pure types, i.e. the components of tuples and the fields of records are *immutable*. In other words, there is no way to assign a field.

4.3.2 Sum Types

We can define new sum types in a functional programming style, with the form of declaration

type t =
$| \ C_1(\tau_{1,1}, \ldots, \tau_{1,n_1})$
$| \ \cdots$
$| \ C_k(\tau_{k,1}, \ldots, \tau_{k,n_k})$

where the $C_1, \ldots, C_k$ are the *constructors*.

Example 4.3.1 *The type of cards in a standard 54-card game can be defined as*

```
type suit = Spade | Heart | Diamond | Club
type value = Ace | King | Queen | Jack | Number(int)
type card = Normal(suit,value) | Joker
```

Associated to sum types is the definition by pattern-matching, with the form

match $expression$ with
$| \ C_1(x_{1,1}, \ldots, x_{1,n_1}) \rightarrow expression_1$
$| \ \cdots$
$| \ C_k(x_{k,1}, \ldots, x_{k,n_k}) \rightarrow expression_k$
end

where each $x_{i,j}$ is a variable which is bound in $expression_i$. For convenience, one may use *extended pattern-matching* [8], where in the patterns an argument of a constructor may itself be a pattern. One may also use the wildcard character _. As in Ocaml or Why3, we distinguish between constructors and variables by enforcing a capital letter as the first character of a constructor.

[1] In Why3 the notation for records uses extra bars as in {| .. |}.

Example 4.3.2 *Here is a function that returns the score of a given card in an imaginary game.*

```
function score (c:card,trump:suit) : int =
 match c with
 | Joker → 100
 | Normal(Ace,s) → if s=trump then 50 else 20
 | Normal(Number(n),s) → if s=trump then 50 else n
 | Normal(v,_) →
  match v with
  | King | Queen → 40
  | Jack → 30
  | _ → 0
  end
 end
```

4.3.3 Recursive Sum Types

Sum type definitions can be recursive. Recursive definitions of functions or predicates are allowed if recursive calls are on *structurally smaller* arguments, which corresponds to the notion of *primitive recursion* on such a sum type.

Example 4.3.3 *Here is a definition of binary trees holding integers at each node, and a function giving the sum of the elements of a tree.*

```
type tree = Leaf | Node(tree,int,tree)
function sum (t:tree) : int =
 match t with
 | Leaf → 0
 | Node(l,x,r) → sum(l) + x + sum(r)
```

The recursion is allowed since arguments l *and* r *are subtrees of the input.*

4.3.4 Polymorphic Types

All the constructs for type definitions above can be parameterized, i.e. the type is given one or several type variables as parameters, as in

```
type vector α = (α,α,α)
type tagged α = { tag : int ; content : α }
type option α = None | Some(α)
type tree α β = Leaf(α) | Node(tree α β, β, tree α β)
```

4.3.5 Example of Lists

Lists, or finite sequences, are very common data structures. The type of lists can be modeled by a polymorphic, recursive sum type as

```
type list α = Nil | Cons α (list α)
```

Classical functions on lists can be defined by primitive recursion, such as

```
function append(l1:list α,l2:list α) : list α =
  match l1 with
  | Nil → l2
  | Cons(x,l) → Cons(x,append(l,l2))
  end

function length(l:list α) : int =
  match l with
  | Nil → 0
  | Cons(x,r) → 1 + length(r)
  end

function rev(l:list α) : list α =
  match l with
  | Nil → Nil
  | Cons(x,r) → append(rev(r),Cons(x,Nil))
  end
```

When proving programs operating on lists, and on recursive sum types in general like binary trees above, it is very common that some properties are needed, such as the properties

```
lemma append_nil: forall l:list α. append(l,Nil) = l
lemma append_assoc:
  forall l1 l2 l3:list α. append(append(l1,l2),l3) = append(l1,append(l2,l3))
```

Such lemmas can be stated as this in the Why3 tool. They must be proved by induction, meaning that they are out of reach to SMT solvers. In the Why3 tool, an interactive theorem prover like Coq can be used to prove them. See Exercise 4.7.6 at end of this chapter.

4.4 Abstract declarations, Axiomatizations

4.4.1 Axiomatic Declarations of Functions and Predicates

Instead of *defining* new logic functions, is it sometimes handy to model *abstract* functions, in the sense that only their profile is declared, and their meaning is partially specified using axioms.

Such an abstract declarations has the form

function $f(\tau, \ldots, \tau_n) : \tau$
predicate $p(\tau, \ldots, \tau_n)$

and is accompanied with *axioms*

axiom id : $formula$

Such a declaration specifies that f, resp. p, is *any symbol satisfying the axioms*.

All functions in the logic are assumed total, so that there is no possible "runtime error" in the specifications themselves. Axiomatizations provide a way to specify partial functions, in the sense that the axioms do not specify their results outside their domains.

Example 4.4.1 *A typical example is the division which can be axiomatized as*

```
function div(real,real):real
axiom mul_div: forall x,y. y≠0.0 → div(x,y)*y = x
```

An expression like div(1.0,0.0) *is a valid expression, but whose value is not specified.*

Example 4.4.2 *The factorial function can be axiomatized as follows.*

function fact(int):int
axiom fact0: fact(0) = 1
axiom factn: **forall** n:int. $n \geq 1 \rightarrow$ fact(n) = n $*$ fact(n$-$1)

The values of fact(n) *for* n *negative are not specified.*

Example 4.4.3 *Taking respectively the head and the tail of a list is undefined when applied on an empty list. These two functions cannot be defined directly by pattern-matching, but they can be axiomatized as follows.*

function head(l:list α):α
axiom head_cons: **forall** x:α, l:list α. head(Cons(x,l)) = x
function tail(l:list α):list α
axiom tail_cons: **forall** x:α, l:list α. tail(Cons(x,l)) = l

Again the values of head(Nil) *and* tail(Nil) *are not specified.*

4.4.2 A Warning about Consistency Issues

We repeat that the meaning of an axiomatization is that the abstract functions and predicates indeed denote any function and predicate satisfying the given axioms. This does not cause any problem of soundness but a potential problem of *inconsistency*: the axioms may be contradictory in the sense that no function could satisfy all axioms at the same time.

A toy example of inconsistent axiomatization is

function f(int):int
axiom f0: f(0) = 1
axiom f1: f(0) = 2

A less toy example is

function div(real,real):real
axiom mul_div: **forall** x,y. div(x,y)$*$y = x

from which we get div(1.0,0.0)$*$0.0 = 1.0 which contradicts **forall** x:real. x$*$0.0 = 0.0.

Typically, in such an context, an automated theorem prover will exploit the inconsistency by proving any goal it is given, including false. In other words, special care must be taken when designing axiomatizations, to avoid introducing inconsistencies.

4.4.3 Axiomatic type declarations

Not only functions but also data types can be declared abstractly, by a declaration of the form

type t

For example, one may introduce a type of colors with

type color
function blue : color
function red : color
axiom distinct: red $\neq$ blue

Unlike sum types, such a type is *open* in the sense that there may exist other values of this type than the constant introduced, e.g. one could introduce later on

function green : color
function rgb(r:real,g:real,b:real) : color
axiom green_rgb: green = rgb(0.0,1.0,0.0)

4.4.4 Polymorphic Types

Abstract types may also be declared as polymorphic, with a declaration of the form

type $t\ \alpha_1 \cdots \alpha_k$

where $\alpha_1 \cdots \alpha_k$ are *type parameters*

A classical example is the type of sets of elements of some type, which can axiomatized as follows.

type set α

function empty : set α

function single(α):set α

function union(set α, set α):set α
axiom union_assoc: **forall** x y z:set α. union(union(x,y),z) = union(x,union(y,z))
axiom union_comm: **forall** x y:set α. union(x,y) = union(y,x)

predicate mem(α,set α)
axiom mem_empty: **forall** x:α. $\neg$ mem(x,empty)
axiom mem_single: **forall** x y:α. mem(x,single(y)) $\leftrightarrow$ x=y
axiom mem_union: **forall** x:α, y z:set α. mem(x,union(y,z)) $\leftrightarrow$ mem(x,y) or mem(x,z)

4.5 Inductive Predicates

Defining a predicate *inductively* is a notion that originally comes from the theory of automated reasoning based on resolution. A *Horn clause* is a first order clause, i.e. a disjunction of literals, with at most one positive literal. For this form of clauses, there exist efficient reasoning methods, which gave birth to the Prolog programming language.

An inductive definition of a predicate is a kind of axiomatization where the axioms are Horn clauses. They appear in proof environments like Coq or PVS. It has the form

inductive $p(\tau_1, \ldots, \tau_n)$:
| id_1: $clause_1$
...
| id_k: $clause_k$

where each clause has the form

forall $\vec{x}$. $hyp \rightarrow p(e_1, \ldots, e_n)$

and p occurs only positively in hyp (thus it is a Horn clause). Another point of view on such definitions is that they define uniquely the predicate, satisfying the clauses, that is true the less often. Such a "smallest" predicate exists, it is the smallest fix-point of the set of clauses.

A typical case when an inductive definition comes naturally is the definition of the transitive closure of a relation.

```
predicate r(x:t,y:t) = ...
inductive r_star(t,t) =
| empty: forall x:t. r_star(x,x)
| single: forall x y:t. r(x,y) → r_star(x,y)
| trans: forall x y z:t. r_star(x,y) ∧ r_star(y,z) → r_star(x,z)
```

The definition should informally read as "r_star is the smallest reflexive and transitive relation that includes r".

4.6 Programs with Arrays

The most common data type appearing in programming languages is the type of arrays. Proving programs with arrays can be done by considering arrays as references to *purely functional arrays*, also called *maps*. Purely functional arrays are just one particular case of a logic datatype that can be axiomatized in first-order logic. The typical axiomatization is as follows.

```
type map α
function select(map α, int) : α
function store (map α, int, α) : map α
axiom select_store_eq:
  forall a:map α, i:int, v:α. select(store(a,i,v),i) = v
axiom select_store_neq:
  forall a:map α, i,j:int, v:α. i ≠ j → select(store(a,i,v),j) = select(a,j)
```

The type map α denotes maps indexed by integers holding values of type α. In this axiomatization, the indices are unbounded: the case of bounded arrays will be addressed later. The function select allows to access the i-th element of a map, that is select(a,i) models the usual notation a[i]. The function store denotes the *functional update* of a map, that is store(a,i,v) denotes a new map whose values are identical to those of a, except at index i where the value is the given v. The two axioms above axiomatize this informal definition of store.

It should be noticed that the theory of maps as above was one of the very first "built-in" theory considered into SMT solvers.

In a program, an array variable is seen as a reference to such a map. The standard assignment operation of the form a[i] := e is interpreted as a := store a i e.

Remind that our IMP language does not allow references to references. Hence, with the approach of arrays above there is no way to build an array of references. In particular, this representation of arrays by references to maps cannot model arrays of arrays were some *sharing* of sub-arrays is allowed. Sharing sub-arrays raises the so-called *aliasing* issues and will be discussed in a next chapter.

An toy example of a program on an array is as follows.

```
val a: ref (map int)

procedure test() :
  writes a
  ensures select(a,0) = 13
body:
  a := store(a,0,13);    (* a[0] := 13 *)
  a := store(a,1,42)     (* a[1] := 42 *)
```

The proof using WP proceeds as follows:

$$
\begin{aligned}
&\mathrm{WP}((a := store(a,0,13); a := store(a,1,42)), select(a,0) = 13)\\
&= \mathrm{WP}(a := store(a,0,13), \mathrm{WP}(a := store(a,1,42)), select(a,0) = 13))\\
&= \mathrm{WP}(a := store(a,0,13), select(store(a,1,42),0) = 13)\\
&= \mathrm{WP}(a := store(a,0,13), select(a,0) = 13)\\
&\quad \text{(thanks to axiom select\_store\_neq)}\\
&= select(store(a,0,13),0) = 13\\
&= 13 = 13\\
&\quad \text{(thanks to axiom select\_store\_eq)}
\end{aligned}
$$

4.7 Exercises

Exercise 4.7.1 *Find appropriate precondition, postcondition, loop invariant, and variant, for the following program, using the axiomatization of Example 4.4.2.*

```
val res : ref int

procedure fact_imp (x:int) :
  requires ?
  writes ?
  ensures ?
body:
  let y = ref 0 in
  res := 1;
  while y < x do y := !y + 1; res := !res * !y done
```

Prove your program using WP.

Exercise 4.7.2 *A procedure that permutes the contents of cells i and j in an array a is as follows.*

```
val a: ref (map int)

procedure swap(i:int,j:int) :
  writes a
  ensures select(a,i) = select(a@Old,j) ∧ select(a,j) = select(a@Old,i) ∧
          forall k:int. k ≠ i ∧ k ≠ j → select(a,k) = select(a@Old,k)
body:
  let tmp = select(a,i) in       (* tmp := a[i] *)
  a := store(a,i,select(a,j));   (* a[i] := a[j] *)
  a := store(a,j,tmp)            (* a[j] := tmp *)
```

- *Prove swap using WP*
- *Prove the procedure*

```
procedure test () :
  requires
    select(a,0) = 13 ∧ select(a,1) = 42 ∧ select(a,2) = 64
  ensures
    select(a,0) = 64 ∧ select(a,1) = 42 ∧ select(a,2) = 13
body swap(0,2)
```

Exercise 4.7.3 *Incrementing an array of reals.*

1. *Write a program over 3 variables: an array of reals a, and two integers min and max. The program should increment by 1 all the values of a between min and max*
2. *Specify a post-condition stating that the values between indices min and max are incremented.*
3. *Specify an appropriate loop invariant.*

Exercise 4.7.4 *The goal is to specify and prove algorithms for searching a value in an array. Such a procedure is expected to fit this canvas:*

```
val a: ref(map real)
val idx : ref int

procedure search (n:int, v:real):
  requires 0 ≤ n
  writes idx
  ensures ?
  body ?
```

1. *Propose a suitable postcondition specifying that if v occurs in array a, between indexes 0 and $n-1$, then idx is an index of such an occurrence, otherwise idx is set to -1*
2. *Implement and prove an algorithm of* linear search*, that follows the pseudo-code:*

   for each i from 0 to $n-1$: if $a[i] = v$ then $idx := i; exit$

3. *Specify and prove an algorithm of* binary search*, that assumes the array sorted in increasing order, and follows the pseudo-code:*

   low = 0; $high$ = n-1;
   while $low \leq high$: let m be the middle of low and $high$
   if $a[m] = v$ then $idx := m; exit$
   if $a[m] < v$ then continue search between m and $high$
   if $a[m] > v$ then continue search between low and m

Exercise 4.7.5 *The goal is to specify and prove algorithms for sorting an array of reals in increasing order. Such a procedure is expected to fit this canevas:*

```
val a: ref(map real)

procedure sort(n:int):
  requires 0 ≤ n
  writes a
  ensures ?
  body ?
```

1. *Propose a suitable postcondition specifying that at the end of execution:*

- *the array is in increasing order between indexes* 0 *and* $n-1$
- *the array at exit is a permutation of the array at entrance*

It is suggested to use an inductive definition to formalize the permutation property.

2. *Implement and prove an algorithm of* selection sort*, that follows the pseudo-code:*

   for each i from 0 to $n-1$:
   finds the index idx of the minimal element between i and $n-1$;
   swap elements at indexes i and idx

   The use of an auxiliary procedure is recommended.

3. *Implement and prove an algorithm of* insertion sort*, that follows the pseudo-code:*

   for each i from 1 to $n-1$:
   insert element at index i at the right place between indexes 0 and $i-1$

Exercise 4.7.6 *We consider a procedure that computes the reversal of a list, in an efficient way (i.e. in linear time in function of the size of the list).*

```
val l:ref (list int)

procedure rev_append(r:list int)
 variant ?
 writes l
 ensures ?
 body match r with
  | Nil → skip
  | Cons(x,r) → l := Cons(x,l); rev_append(r)

procedure rev(r:list int)
 writes l
 ensures l = rev r
 body ?
```

1. *Fill the ? with appropriate annotations or code.*

2. *Prove the program using WP. Which general lemmas on lists are needed?*

Exercise 4.7.7 *Resolution of a second degree equation.*

- *Write an IMP program that assumes 3 variables* a, b, c *of type real and stores into a variable of type (list real) the set of solutions of the equation* $ax^2 + bx + c$*.*
- *Specify the expected behavior in the post-condition.*
- *Prove the program. You may pose lemmas to state some mathematical results.*

Exercise 4.7.8 *Sum of elements in a list.*

- *Using the standard datatype of polymorphic lists, define a logic function which gives the sum of elements of a list of real numbers.*
- *Assuming 2 global variables l of type* list real *and s of type* real*, write a program that eventually stores in s the sum of elements of l. You may add extra local variables if needed.*
- *Propose a post-condition that specifies the expected behavior of the program.*
- *Propose appropriate loop invariants and variants to allow the proof of the program using* WP.

Example 4.7.9 *Considering polymorphic binary trees defined as*

type tree α = Leaf | Node(tree α,α,tree α)

specify, implement and prove a procedure returning the maximum of a tree of integers.[2]

Bibliography

[1] J.-R. Abrial. *The B-Book, assigning programs to meaning*. Cambridge University Press, 1996.

[2] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: The common algebraic specification language. *Theoretical Computer Science*, 286 (2):153–196, 2002.

[3] M. Barnett, R. DeLine, B. Jacobs, B.-Y. E. Chang, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, 2005.

[4] C. Barrett and C. Tinelli. CVC3. In Damm and Hermanns [11], pages 298–302. URL `ftp://ftp.cs.uiowa.edu/pub/tinelli/papers/BarTin-CAV-07.pdf`.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

[6] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The Alt-Ergo automated theorem prover, 2008. `http://alt-ergo.lri.fr/`.

[7] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011. URL `http://proval.lri.fr/submissions/boogie11.pdf`.

[8] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. *The Why3 platform, version 0.71*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.71 edition, Oct. 2011. `https://gforge.inria.fr/docman/view.php/2990/7635/manual.pdf`.

[9] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005. URL `http://dx.doi.org/10.1007/s10009-004-0167-4`.

[2] Problem 2 of the FoVeOOS verification competition in 2011, `http://foveoos2011.cost-ic0701.org/verification-competition`

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, Aug. 2002.

[11] W. Damm and H. Hermanns, editors. *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, Berlin, Germany, July 2007. Springer.

[12] L. de Moura and N. Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[13] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. North-Holland, 1990.

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

[15] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Damm and Hermanns [11], pages 173–177. URL `http://www.lri.fr/~filliatr/ftp/publis/cav07.pdf`.

[16] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. *Software Engineering with OBJ: Algebraic Specification in Action*, chapter Introducing OBJ*. Kluwers Academic Publishers, 2000.

[17] J. V. Guttag and J. J. Horning. *Larch: languages and tools for formal specification*. Springer, New York, NY, USA, 1993. ISBN 0-387-94006-5.

[18] C. B. Jones. *Systematic software development using VDM (2nd ed.)*. Prentice-Hall, 1990. ISBN 0-13-880733-7.

[19] K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Springer, editor, *LPAR-16*, volume 6355, pages 348–370, 2010.

[20] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, March 2000. ISBN 0136291554. URL `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0136291554`.

[21] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27 (2):356–364, 1980. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322186.322198.

[22] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[23] M. Norrish. Hol 4 kananaskis-4. `http://hol.sourceforge.net/`.

[24] S. Owre and N. Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, Aug. 1997.

[25] S. Ranise and C. Tinelli. The smt-lib format: An initial proposal. In *Proceedings of PDPAR'03*, July 2003.

[26] W. Reif, G. Schnellhorn, and K. Stenzel. Proving system correctness with KIV 3.0. In W. McCune, editor, *14th International Conference on Automated Deduction*, Lecture Notes in Computer Science, pages 69–72, Townsville, North Queensland, Australia, july 1997. Springer.

[27] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, 1989. ISBN 0-13-983768-X.

[28] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2 edition, 2003. ISBN 0321179366.