

Simple Type Theory: Simple Steps Towards a Formal Specification

William M. Farmer Martin v. Mohrenschildt
 Department of Computing and Software
 McMaster University
 1280 Main Street West
 Hamilton, Ontario L8S 4K1
 Canada
 {wmfarmer, mohrens}@mcmaster.ca

Abstract - Engineers, particularly software engineers, need to know how to read and write precise specifications. Specifications are made precise by expressing them in a formal mathematical language. Simple type theory, also as known as higher-order logic, is an excellent educational and practical tool for creating and understanding formal specifications. It provides a better logical foundation for specification than first-order logic and is a better introductory specification language than industrial specification languages like VDM-SL and Z. For these reasons, we recommend that simple type theory be incorporated into the undergraduate engineering curriculum.

like VDM-SL [2] and Z [20].

Expressivity. Simple type theory is a highly expressive specification language as well as a powerful logic. It contains the basic logical tools needed for writing clear and concise formal specifications. They are:

- 1) Propositional connectives.
- 2) Universal and existential quantifiers.
- 3) Application and abstraction operators (e.g., for applying and defining functions, respectively).
- 4) Higher-order objects (such as functions and sets).
- 5) Types.
- 6) A definite description operator (for forming expressions of the form “the unique object x that satisfies the property P ”).

We will illustrate each of these elements later in the paper. It is important to note that first-order logic contains only a few of these elements, namely, propositional connectives, quantifiers, and an operator for applying functions and predicates.

Practicality. Although a textbook formulation of simple type theory is a much more practical specification language than first-order logic, it would be burdensome to write large, complex specifications in it. However, by extending its syntax and semantics in certain ways, simple type theory can be made into an effective specification language for actual use. These extensions do not change simple type in any fundamental way; they just make simple type theory more convenient to employ.

Simple type theory provides a better logical foundation for formal specification than first-order logic. It is a better introductory specification language than the industrial specification languages VDM-SL and Z. And some of the most popular and advanced computer theorem proving systems, including HOL [10], IMPS [7], Isabelle [18], ProofPower [13], and PVS [16], are based on logics that are extensions of simple type theory. For these reasons, we recommend that simple type theory be incorporated into the undergraduate engineering curriculum.

The paper is organized as follows. Section II explains what simple type theory is and introduces a practical version of simple type theory called BESTT. The process of composing

INTRODUCTION

The creation and use of specifications is fundamental to engineering practice. Software engineers—as well as many other engineers—need to know how to read and write precise specifications. Specifications are made precise by expressing them in a formal mathematical language. Such *formal specifications* have many advantages over informal specifications written in natural language because they can be mechanically constructed and analyzed by humans or by software. However, most engineers lack the background required to employ formal specifications, despite the fact that undergraduate engineering programs usually include some exposure to logic and discrete mathematics.

Simple type theory, also as known as *higher-order logic*, is a natural extension of first-order logic which is simple, highly expressive, and practical. (For a full discussion of the virtues of simple type theory, see [5].) It is an excellent educational and practical tool for creating and understanding formal specifications. This is illustrated by simple type theory’s three strengths.

Simplicity. Simple type theory has a very simple syntax and semantics. It contains fewer syntactic categories than first-order logic, and it is based on the same semantic principles as first-order logic. It is therefore not much harder for students to learn than first-order logic. By virtue of its simplicity, it is also much easier to learn than industrial specification languages

a specification is outlined in section III. The main section of the paper, section IV, illustrates how specifications can be written in BESTT. Simple type theory as a specification language is compared with the leading industrial specification languages in section V, and software tools for manipulating specifications written in simple type theory are briefly discussed in section VI. The paper ends with some remarks about our experiences using simple type theory at McMaster in section VII and some concluding remarks in section VIII.

WHAT IS SIMPLE TYPE THEORY?

As we said in the Introduction, simple type theory is a natural extension of first-order logic which is simple, highly expressive, and practical. There are many different formulations of simple type theory, but the most common formulation today is a system due to A. Church [1], known as *Church's type theory*, which includes machinery for applying and specifying functions and a definite description operator. In this paper we will always assume that “simple type theory” means “Church's type theory”.

Simple type theory is the most popular form of type theory. Simple type theory, like other type theories, has two kinds of syntactic objects. *Expressions* denote values including the truth values T (true) and F (false); they do what both terms and formulas do in first-order logic. *Types* denote nonempty sets of values; they are used to restrict the scope of variables, control the formation of expressions, and classify expressions by value. For example, a formula is an expression of type `BOOL` that denotes a truth value.

Unlike some other type theories, simple type theory is a classical, nonconstructive, two-valued logic. It includes strong support for specifying and reasoning with a hierarchy of higher-order functions. (A function is *higher order* if it takes other functions as arguments.) The structure of the hierarchy is inherited from the structure of the types. In contrast to a *set theory* (such as Zermelo-Fraenkel (ZF) set theory), simple type theory is a *function theory* in which functions are also used to represent other kinds of values such as sets and relations.

There are several ways of extending simple type theory so that it is more practical for actual use (see [5]). One example of a practical extension of simple type theory developed at McMaster University is BESTT [4], a Basic Extended Simple Type Theory. BESTT has type variables for forming polymorphic types and expressions as in the HOL logic [10] and built-in machinery for working with tuples, lists (finite sequences), and sets.

Undefinedness arises naturally throughout mathematics, computer science, and engineering. For example, division by 0 is never defined and many computer programs do not terminate on all inputs. Since undefinedness is often unavoidable, a practical logic for specification must have a mechanism for dealing with it. BESTT comes with a *partial semantics* in which functions may be partial and expressions may be undefined (but formulas are always either true or false). This semantics formalizes the *traditional approach to undefinedness* employed

in mathematical practice (see [3], [6]). As a result, undefined applications of partial functions like $1/0$ and undefined definite descriptions like “the unique x such that $x \neq x$ ” can be directly expressed in BESTT (for more examples, see [6]).

THE PROCESS OF COMPOSING A SPECIFICATION

By a specification we mean a document written in a formal language—a language having a precise syntax and semantics—that describes the requirements, interfaces, data structures, module designs, etc. of a system. While there are philosophical discussions about how formal or informal a specification should be, we think that it is essential that engineering students, particularly software engineering students, are capable of making precise and unambiguous statements about small systems.

The process of composing a specification for a system consists of seven steps. The first three are for all systems, but the last four are just for systems that keep a state or memory.

- 1) *Define the types.* The first step is to define the types of values—quantities, data, objects, inputs, outputs, etc.—with which the system is concerned. If this step is done correctly, the remaining steps are often straightforward.
- 2) *Declare the constants.* The constants are the primitives of a language L for describing the values and making assertions about them. For example, the constants may include $0, 1, -, +, *$ for talking about integers.
- 3) *State the axioms.* The axioms are a set Γ of statements in L that are assumed to be true about the values. The language L and set Γ of axioms form a theory that specifies the values. For example, the axioms may include a statement that says 0 is the identity with respect to $+$ over the integers.
- 4) *Declare the state variables.* If the system we are specifying keeps a state or memory, the state variables are special constants that represent components of the system's state. For example, the two state variables for a stack might be an array that holds the stack elements and a natural number that holds the height of the stack.
- 5) *State the invariants.* The invariants of the system are statements involving the state variables that are assumed to remain true in all states of the system. In the stack example above, the statement that the height is between 0 and the length of the array would be an invariant.
- 6) *Declare the operations.* The operations are special constants that effectively read or write state variables. They may also take inputs and return outputs. They represent how the system interacts with its state. In the stack example, `top`, `pop`, and `push` would be operations.
- 7) *Specify the operations.* The operations are either defined as functions that map states to states or are specified by the properties they satisfy, for example, using pre- and postconditions. In the stack example, the `top` operation could be defined as a partial function that maps a nonempty stack of elements to the top element of the stack.

In practice, the specification writer usually does not do the steps in an entirely sequential manner but instead goes back and forth between the steps until a complete specification is produced.

WRITING SPECIFICATIONS IN BESTT

Before a student is able to compose a specification using BESTT, or any other specification language, he or she must become familiar with logic, and in our case with higher-order logic. The student also needs to have acquired basic knowledge in discrete mathematics, data structures, algorithms, and finite state machines. Due to space limitations, we can only give a short introduction to the language of BESTT. For a complete introduction the reader is referred to [4]. (The notation we will use in this paper is slightly different than that in [4].) We would like to point out that the language of BESTT is very compact, containing only a small number of language elements compared to other specification languages such as VDM-SL and Z. A detailed comparison between BESTT and other specification languages is given in section V. The reader familiar with the ML [19] functional programming language or the PVS the specification language will notice many similarities between BESTT and these languages.

Following the process of composing a specification presented in section III, a specification in BESTT can contain five types of specification elements:

- 1) Type definitions.
- 2) Declarations of constants, state variables, and operations.
- 3) Axioms.
- 4) Invariants.
- 5) Definitions or specifications of operations.

Types form the foundation of BESTT. We assume the atomic types include `BOOL`, `INT`, `FLOAT`, `CHAR`, `STRING`, and `UNIT`. Let α and β be types. Compound types are constructed using two binary type constructors: $\alpha \rightarrow \beta$ is the type of functions from α to β and $\alpha * \beta$ is the product type of α and β . For convenience, we allow the usage of records types, e.g., $(x:\text{INT}, y:\text{INT})$, which are essentially product types with named components. Additional compound types are constructed using two unary constructors: `set(α)` and `list(α)` are the types of a set and a list of elements of type α , respectively. We will use the alternation constructor `|`, e.g., `Male|Female`, to construct enumerated types. Finally, we will assume that users can define their own type constructors in BESTT, e.g., `order(α) = ($\alpha * \alpha$) \rightarrow BOOL, including recursive type constructors.`

A type definition is given using the key word `type` followed by an identifier naming that type, possibly a list of type variables for type constructors, and then the type. BESTT allows the definition of abstract types, i.e., types that are not further defined. The syntax we use for type declarations is identical to that used in the language OCaml [14] (a dialect of ML). For example,

```
type Date
type Gender = Male|Female
type Person = STRING * Date * Gender
```

defines the abstract type `Date`, the type `Gender` as an enumeration of constants, and the type `Person` as a product type of `String`, `Date`, and `Gender`. We like to point out that at any time point we could further specify the type `Date` without changing any of the definitions depending on it.

Declarations are used to declare constants, state variables, and operations. A declaration consists of the key word `decl`, an identifier, and a type. For example,

```
decl data_set : set(Person)
decl get_date : Person  $\rightarrow$  Date
```

declare the state variable `data_set` and the operation `get_date`.

Expressions in BESTT are formed from variables and declared constants, state variables, and operations using standard expression constructors. The expression constructors include an if-then-else constructor and constructors to build and access lists such as $\langle a_1, \dots, a_n \rangle$. Sets can be constructed by *enumeration*, e.g., $\{a_1, \dots, a_n\}$, or by *set abstraction*, e.g., $(S a : \alpha . p(a))$ where p is a predicate over the elements of type α . Individual objects can be described using the construct of *definite description*. For example, the definite description $(I a : \alpha . p(a))$, where again p is a predicate over the elements of type α , denotes the unique element that satisfies p if there is such an element and is undefined otherwise. See [4] for a complete list of the expression constructors in BESTT.

Formulas, i.e., expressions of type `BOOL`, are formed using the constants `T`, `F`, representing true and false, the standard logical connectors $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ and the quantifiers \forall, \exists . BESTT also provides the definedness operator \downarrow such that $E \downarrow$ asserts that the expression E is defined.

An invariant is a formula involving state variables that is assumed to be true. Logically, an invariant is just a special kind of axiom. Invariants are stated using the key word `inv` followed by the name and formula of the invariant. For example, the invariant

```
inv unique_birthday =
 $\forall (n, d, g), (n', d', g') : \text{data\_set} .$ 
 $n = n' \rightarrow d = d'$ 
```

states that, for any two objects of type `Person` associated with the variable `data_set`, if the two strings are identical then so are the two birthdays.

BESTT provides the special expression constructor that is used to define a function by means of what is called lambda abstraction in logic. Definitions associate a previously declared identifier with an expression. The definition

```
def get_date = (n, d, g)  $\mapsto$  d
```

defines the operation `get_date` by stating that, given an object of type `Person`, declared to be the product of `STRING`, `Date`, and `Gender`, it returns the element of type `Date`.

Now we can continue with our example by declaring

```
decl find_date : Person  $\rightarrow$  (BOOL * Date)
decl find_people : Date  $\rightarrow$  set(Person)
```

and then make the following definitions to give meaning to

the declared operations:

```
def find_date = n ↦
  ((∃ d : Date, g : Gender . (n, d, g) ∈ data_set),
   (Id : Date . ∃ g : Gender . (n, d, g) ∈ data_set))
```

Note that, by the previously stated invariant `unique_birthday`, the definite description is defined exactly if the first component in the pair is true.

```
def find_people = d ↦
  (S (n, d', g) : Person .
   (n, d', g) ∈ data_set ∧ d = d')
```

`find_people` returns the set of objects of type `Person` that are “born on some given date”.

Summarizing, we have given the specification of a simple system that stores people together with their unique birthdays, and allows us to determine the birthday of a person and the set of all the people born on any particular date.

Higher-order constructs are very useful if one aims to specify abstract concepts such as sorting. Using the type constructor `order(α)`, as defined above, we can declare an operation `sorter` as follows:

```
decl sorter : order(α) → (list(α) → list(α))
```

We can then define the operation `sorter` by definition description:

```
def sorter =
  Is : order(α) → (list(α) → list(α)) .
  ∀ o : order(α), a, b : list(α) .
  s(o)(a) = b ⇔
  (∀ i : INT . 0 ≤ i < |b| - 1 ⇒ o(b[i], b[i + 1]))
  ∧ list_equiv(a, b)
```

where `list_equiv(a, b)` is predicate of type `(list(α) * list(α)) → BOOL` that is true if the sequences `a` and `b` contain the same elements but in possibly a different ordering.

Specifications are often intended to describe dynamic models in which the state of the model can be changed by the specification’s operations, as done in VDM-SL and Z. In this case, we speak about traces of states where a state q is a list of values associated with the specification’s state variables. Since an operation o effectively reads and writes state variables, it can be viewed as a state transition function that moves a state q to a state \tilde{q} , which is written as $\delta(q, o) = \tilde{q}$. Let Q be the state space, i.e., the set of all possible states, and O be the set of operations. We introduce the notation $q \models A$ to denote that the formula A holds in the state q . Then $q \models A'$ means that, for each $o \in O$, if $\delta(q, o) = \tilde{q}$, then $\tilde{q} \models A$, i.e., that A holds in each possible next state. $q \models (A \Rightarrow B)$ means that $q \models A$ implies $q \models B$. There are similar definitions for the other propositional connectives.

An invariant is now a formula A such that

$$q_0 \models A \wedge \forall q : Q . q \models (A \Rightarrow A')$$

where q_0 is the initial state. With this the student is introduced to model checking as a tool to verify properties of specifica-

tions. Traces allow us to define the notion of an *event* as done in [9] and [12]: $@T(A)$ denotes the time points in a trace $\langle q_0, q_1, \dots \rangle$ at which $q_i \models (\neg A \wedge A')$, and $@T(A)$ WHEN D denotes the time points at which $q_i \models \neg A \wedge A' \wedge D \wedge D'$, i.e., at which A changes from false to true and D remains true. (This could also be defined using previous states instead of next states.)

To illustrate the usage of events we give a small specification that defines an event that occurs if a user clicks the mouse in a button (fixed area) on the screen. We need the following type declarations:

```
type mouse_button = Up|Down
type mouse_pos = (x:INT, y:INT)
```

Now we declare the state of the mouse using a state variable:

```
decl mouse : (b: mouse_button, pos: mouse_pos)
```

We need a predicate that is true if the mouse is pointing to the area of interest:

```
decl in_area : mouse_pos → BOOL
```

The predicate `in_area` can easily be defined using that the area is located at position (x_0, y_0) with a width w and height h as

```
def in_area = (x, y) ↦
  (x_0 ≤ x ≤ x_0 + w) ∧ (y_0 ≤ y ≤ y_0 + h)
```

Now we can specify the events of interest, first that the mouse button was pressed while the mouse pointed to the area:

```
def B1 = @T(mouse.b = Down)
  WHEN in_area(mouse.pos.x, mouse.pos.y)
```

Note that these events do not happen if we move the mouse with the button down over the area without pressing it down within the area. Further we need the events when the mouse button was released while the mouse was within the area and while it was released outside the area:

```
def B2 = @T(mouse.b = Up)
  WHEN in_area(mouse.pos.x, mouse.pos.y)
def B3 = @T(mouse.b = Up)
  WHEN ¬in_area(mouse.pos.x, mouse.pos.y)
```

If a B1 event is followed by a B2 event then the mouse was clicked and released within the area, but if it is followed by a B3 event then the mouse was clicked but not released within the area.

COMPARISON WITH OTHER SPECIFICATION LANGUAGES

Instructors introducing students to formal specification can choose from a variety of different specification languages. In general, one can distinguish two classes of specification languages: *model oriented* and *property oriented*. Model-oriented languages, such as the PVS [16] specification language; UML [11], the Universal Modeling Language; VDM-SL [2], the specification language of the Vienna Development Method (VDM); and Z [20] (called Z after Zermelo-Fraenkel set theory), are languages that describe the operations by defining

how they affect the models (state space) they work on. This includes the notion of pre- and postconditions and the different kinds of specification languages using finite state machines. Property-oriented languages, often called algebraic languages, make abstract statements about the properties of the operations, which could be satisfied by many models. The commonly used example is the stack. A property-oriented language would state that $\text{pop}(\text{push}(s, t)) = s$ and $\text{top}(\text{push}(s, t)) = t$, while a model-oriented language would first declare that a stack has a state represented as a list of objects and then describe how top , pop , and push affect this state.

In contrast to Z, BESTT allows expressions to be undefined. BESTT is not a three-valued logic; the logic provides only the usual two boolean values, and formulas are always defined. We think that this approach is very natural, e.g., $\text{top}(\text{empty_stack})$ is undefined. Specification languages cannot ignore occurrences of undefinedness. They can either deal with them indirectly in various ways or directly as BESTT does, by using artificially constructed types with undefined values, or by introducing the notion of an “exception”.

The languages UML and Z use two dimensional graphical constructs to assist the composer and reader of specifications. In our software design courses, we found that using BESTT within Parnas Tables [17] increased the readability of specifications in a similar way.

To illustrate the difference in syntax we define the same structure, a simple binary tree in BESTT, VDM-SL, and Z: The BESTT specification is

```
type Btree( $\alpha$ ) = Empty|Node(Btree( $\alpha$ ) *  $\alpha$  * Btree( $\alpha$ ))
```

where the type $\text{Btree}(\alpha)$ is parameterized with the type of the content of the nodes. This line can be directly sent to an OCaml interpreter or compiler. Since VDM-SL and Z do not support type variables, we cannot specify a parameterized type, but we have to construct our trees over some fixed type, e.g., INT. The binary tree specification in VDM-SL is given by

```
binnode :: left : Btree
          value : INT
          right : Btree
Btree = [binnode]
```

and in Z the same specification is

```
Btree ::= nil|binnode << Btree  $\times$  INT  $\times$  Btree >>
```

The specifications in BESTT and Z look very similar, but the VDM-SL specification is different because Btree is not actually a type in VDM-SL, only binnode is. We found, confirmed by our experiences teaching formal specification to students, that the BESTT notation is more natural.

SOME REMARKS ABOUT SOFTWARE TOOLS

Simple type theory can be used as a specification language without the aid of software tools—especially for educational purposes. However, as one considers specifications of greater size and complexity, it becomes increasingly more difficult

to express and analyze them in simple type theory without support from software tools. In short, practical specification using simple type theory (or any other formal specification language) requires tool support.

There are three basic kinds of software tools for formal specification:

- 1) *Tools for constructing specifications.* These tools help the user to find syntactic mistakes in specifications and to combine smaller specifications into larger ones. They include *type checkers* for checking whether a specification is type correct and *type inferers* for determining the types of the specification components. The latter are included in implementations of the ML programming language.
- 2) *Tools for analyzing specifications using symbolic computation.* There are a wide range of tools of this kind. These include *model checkers* for determining whether a property holds in every state of a specified system, *computer algebra systems* for simplifying algebraic expressions, *decision procedures* that can automatically decide whether certain kinds of conjectures are true or false, systems for finding counterexamples to conjectures, and various kinds of systems for visualizing the models that satisfy a specification.
- 3) *Tools for analyzing specifications using formal deduction.* These include *automatic theorem provers* like Otter [15] that attempt to automatically find a proof for a conjecture and *proof development systems* like PVS with which the user and the system interactively construct a proof of a conjecture.

Specification systems of the future will contain an integrated set of tools of all three kinds described above. They will be effectively “laboratories” in which to construct and analyze specifications and in which symbolic computation and formal deduction are fully integrated (e.g., see [8]).

EXPERIENCES AT MCMASTER UNIVERSITY

By teaching introductory courses in software engineering to undergraduate and graduate students in software engineering, computer engineering, and electrical engineering, we have become very familiar with the problems students have in understanding formal specification techniques and languages. Our experiences are what prompted us to develop BESTT and our approach to teaching specification.

Our approach has been incorporated in the B.Eng. program in software engineering at McMaster University. Designed in 1999 and revised in 2003, this program is accredited by the Canadian Engineering Accreditation Board (CEAB). The first year is a common year to all engineering students. In the 2003 revision we decided to postpone the first introductory course in software engineering to the second term in the second year. Students first have to pass three second-year first-term courses: one course in logic, a separate course in discrete mathematics, and a course that introduces them to the fundamentals of programming as a continuation of their first-year programming course. In their first software

engineering course, students learn to specify and design small systems or modules. A second course in software engineering then exposes students to the paradigms and methodologies of designing large scale systems (see [21]). We do not use any of the “popular” specification languages and methods such as B, UML, VDM-SL, and Z, but instead we show our students how to write precise specifications using BESTT.

For undergraduate computer and electrical engineering students, we have taught an introduction to BESTT as part of their first software engineering course in the third year. At the graduate level in software engineering we have two core courses, one in logic in the first term and one in software design in the second term.

We have found that the students taking the logic and the software engineering courses in which BESTT is used are able to quickly understand BESTT and to write specifications in it. The syntax and semantics of BESTT have posed no special problems for them. Most of their difficulties, in fact, come from the underlying principles of BESTT, such as quantification and the distinction between syntax and semantics, which are the same underlying principles of first-order logic. One nice benefit of using BESTT is that students can quickly memorize its language since it is so simple.

CONCLUSION

In this paper we have argued that simple type theory is an excellent educational and practical tool for creating and understanding formal specifications and that simple type theory provides a better foundation for specification than first-order logic. We therefore recommend that simple type theory be incorporated into the undergraduate engineering curriculum in three ways.

First, the traditional two-logic sequence taught in logic and discrete mathematics courses should be replaced with a three-logic sequence—*propositional logic, first-order logic, simple type theory*. This is feasible since nearly all the basic ideas and principles of simple type theory are already found in first-order logic (the notion of a type is the most notable exception). Students would be able to move onto simple type theory soon after being introduced to first-order logic. The three-logic sequence could be taught in either a one-term logic course or a two-term discrete mathematics course.

Second, the techniques for constructing and employing specifications introduced in the other courses in the curriculum should be based on a common foundation derived from simple type theory, and the students should routinely read and write formal specifications expressed in simple type theory. This will give the students a more unified understanding of specification and will help them become proficient and comfortable working with specifications and using logic for practical purposes.

Third, the students should be introduced to state-of-the-art tools for constructing and analyzing formal specifications. They should learn what are the capabilities of different kinds of tools and what are the limitations of specification tools in general.

REFERENCES

- [1] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
- [2] J. Dawes. *The VDM-SL*. Pitman/UCL Press, 1991.
- [3] W. M. Farmer. Reasoning about partial functions with the aid of a computer. *Erkenntnis*, 43:279–294, 1995.
- [4] W. M. Farmer. A Basic Extended Simple Type Theory. SQRL Report No. 14, McMaster University, 2003.
- [5] W. M. Farmer. The seven virtues of simple type theory. SQRL Report No. 18, McMaster University, 2003.
- [6] W. M. Farmer. Formalizing undefinedness arising in calculus. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning—IJCAR 2004*, Lecture Notes in Computer Science. Springer-Verlag, 2004. Forthcoming.
- [7] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.
- [8] W. M. Farmer and M. von Mohrenschildt. An overview of a formal framework for managing mathematics. *Annals of Mathematics and Artificial Intelligence*, 38:165–191, 2003.
- [9] S. R. Faulk and C. L. Heitmeyer. The SCR approach to requirements specification and analysis. In *3rd IEEE International Symposium on Requirements Engineering (RE'97)*, page 263, Annapolis, Maryland, 1997. IEEE Computer Society.
- [10] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
- [11] Object Management Group. UML resource page. On Web, 2004. Available at <http://www.omg.org/uml/>.
- [12] C. L. Heitmeyer, J. Kirby, and B. G. Labaw. The SCR method for formally specifying, verifying, and validating requirements: Tool support. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 610–611, Boston, Massachusetts, 1997. Springer-Verlag.
- [13] Lemma 1 Ltd. *ProofPower: Description*, 2000. Available at <http://www.lemma-one.com/ProofPower/doc/doc.html>.
- [14] X. Leroy. *The Objective Caml System*, 2003. Available at <http://caml.inria.fr/ocaml/htmlman/index.html>.
- [15] W. McCune. OTTER 2.0. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 663–664. Springer-Verlag, 1990.
- [16] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification: 8th International Conference, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, 1996.
- [17] D. L. Parnas, J. Madey, and M. Iglewski. Precise documentation of well-structured programs. *IEEE Transactions on Software Engineering*, 20:948–976, 1994.
- [18] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
- [19] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996.
- [20] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
- [21] M. von Mohrenschildt and D. K. Peters. The Draw-Bot: A project for teaching software engineering. In *1998 Frontiers in Education Conference*, pages 1022–1027, Tempe, Arizona, 1998. IEEE.