11-18-1994

The Direct Execution of SPECS-C++: A Model-Based Specification Language for C++ Classes

Tim Wahls
Iowa State University

Albert L. Baker
Iowa State University

Gary T. Leavens
Iowa State University

The Direct Execution of SPECS-C++: A Model-Based Specification Language for C++ Classes

TR94-02b

Tim Wahls, Albert L. Baker and Gary T. Leavens

November 18, 1994

Iowa State University of Science and Technology
Department of Computer Science
226 Atanasoff
Ames, IA 50011

The Direct Execution of SPECS-C++: A Model-Based Specification Language for C++ Classes

Tim Wahls, Albert L. Baker, and Gary T. Leavens

TR #94-02b
March 1994

Categories and Subject Descriptors:
D.2.1 [**Software Engineering**] Requirements/Specifications — *languages*; D.2.2 [**Software Engineering**] Tools and Techniques — *computer-aided software engineering (CASE)*; D.2.m [**Software Engineering**] Miscellaneous — *rapid prototyping*
General Terms: Specification, Design, Prototyping, Formal Semantics
Additional Key Words and Phrases: C++, Standard ML, executable specification, operational semantics, specification language semantics

Submitted to Formal Methods Europe '94.

The Direct Execution of SPECS-C++: A Model-Based Specification Language for C++ Classes

Tim Wahls[*], Albert L. Baker, and Gary T. Leavens

November 18, 1994

Abstract

Executable specification languages may be the key to more widespread use of formal methods in software production. However, the expressiveness of executable specification languages is typically much less than that of non-executable specification languages such as VDM or Z. Thus, specifiers are forced to work at a lower level of abstraction to gain the advantage of executability. Additionally, specifications are typically made executable by translating them to a programming language, so errors in the specification can only be detected as errors in the resulting code. This paper presents a technique for directly executing specifications written in SPECS-C++, a model-based specification language for C++ classes. As SPECS-C++ has much in common with the implicit subset of VDM, this technique is equally applicable to implicit VDM specifications. Standard ML code for the interpreter and the example used in the paper appear in the appendices.

1 Introduction

One barrier preventing formal methods from playing a larger role in industrial software development is the fear that the benefits of using formal methods are not worth the costs. Executable formal specifications can help in overcoming this fear, as an executable specification yields an immediate prototype of the final system. Those who already use formal specifications also benefit, because executing a specification allows debugging it in the same way that programs are debugged. In addition, executable specifications ease incremental system integration and testing, as the specification (as a prototype) can take the place of parts of the final system that are yet to be implemented.

Thus it is not surprising that many executable specification languages have been developed. Unfortunately, these languages almost always force the specifier to work at a much lower level of abstraction than is normally employed in writing specifications in non-executable model-based specification languages, such as VDM [Jon90] and Z [Hay87, Spi88, Spi89, Spi92]. Typically, either the specifier must provide the algorithms needed for executing the specification [LL91, BL90, ZS86, LB89], or execution is based on translating the specification to Prolog [TC89, WE92, DKC90]. In the former case, the specifier is forced to work in an impoverished specification language that is often almost indistinguishable from modern functional programming languages, and the algorithmic information that must be provided is a potential source of implementation bias [Jon90]. If the specification is translated to Prolog, the person using the executable specification is exposed to the inefficiency and non-logical features of Prolog (cut, dependence on order of clauses). Such

[*]Wahls's work is supported by a fellowship provided by IBM Rochester. Leavens's work is supported in part by the National Science Foundation under Grant CCR-9108654.

translations also usually entail poor reporting of errors. These problems are magnified when validating specifications, as specification errors show up in the Prolog code, and the validator must first find the error in the context of the Prolog code, and then find the corresponding error in the specification.

A number of existing executable specification languages/techniques allow the execution of implicit specifications — that is, specifications that do not explicitly provide the necessary algorithms. We first briefly describe several of these languages, and then discuss some of the limitations that these languages share.

The EPROL specification language [HI88, HI86] makes a fairly large subset of VDM executable by compiling specifications to LISP. Even some specifications that use quantifiers are executable. EPROL is part of the EPROS system, which provides tool support for evolutionary and functionality prototyping and for building the user interface to the prototype. Similarly, the *me too* specification language [Hen86] executes much the same subset of VDM by embedding specifications in LISP.

The UK's National Physical Laboratory has developed a syntax-directed editor for entering VDM specifications with a mode called *SMLVIEW* [O'N92a, O'N92b] that displays the VDM specification entered compiled into Standard ML (SML) [Pau91] code. The translation scheme is apparently fairly direct, so the VDM specification must be somewhat close to an SML program for this translation to work, and the subset of VDM that can be translated is relatively small. Additionally, some VDM features that are in principle executable, such as set comprehensions[1], are not implemented.

As both EPROL and *SMLVIEW* involve translation to a programming language, both force the user wishing to validate specifications into the "debug the code, then debug the specification" mode described previously, especially if the generated code fails to compile (as can happen when using *SMLVIEW*) or causes a run-time error. Any specification error discovered is then reported in the context of the generated code, rather than the context of the specification. As an EPROL specification can be written at a more abstract level than a specification to be executed using *SMLVIEW*, the distance between the specification and the Lisp code generated is relatively large, and the difficulty of finding the flaw in the specification that caused the error in the generated code may be great. As specifications to be executed using *SMLVIEW* must be fairly close to the resulting SML code, this "reverse translation" problem is likely to be less serious. However, with *SMLVIEW*, the user must often hand edit the SML code resulting from the translation to achieve even a syntactically correct SML program. In either case, then, the user is forced into intimate contact with a language that is not directly involved in the software development process – it is neither the specification nor implementation language.

Additionally, all three of the techniques discussed apparently cannot execute VDM-like assertions of the following three forms:

- The only operator that can be used to define post-state values is =. In other words, the assertion $3 \in S'$, where S' represents the post-state value of the set S, is of no help in building a post-state value for S, even though it clearly defines part of that value.

- Post-state values of tuple, set, and sequence types cannot be defined "by parts", i.e. by describing the result of applying tuple, set, and sequence observers to such post-state values. For example, if rationals are modeled as a tuple consisting of *numer* and *denom* fields, then an assertion such as $numer(r') = 3 \wedge denom(r') = 4$ is not executable, even though it clearly defines the post-state value of r.

[1]For example, $\{i + 2 \mid 1 \leq i \leq 5 \wedge i \bmod 2 = 0\}$ is a set comprehension with value $\{4, 6\}$

- A quantified assertion can only be evaluated if it contains no references to post-state values, and so can only be evaluated for its boolean value. Thus, an expression such as $\forall x[x \in S \wedge odd(x) \Rightarrow x \in S']$ can not be used in constructing the value of S', even though it again defines part of that value.

In the case of *SMLVIEW*, these limitations can be seen in the description of the subset of VDM that can be executed (see Appendix J of [O'N92a]). Although the authors have been unable to locate any similar characterization of the subset of VDM that EPROL or *me too* can execute, the example specifications that have been published also exhibit these same limitations.

We present a technique that makes constructive use of these kinds of assertions in constructing post-state values, and that executes the kinds of assertions that can be executed by EPROL, *me too*, and *SMLVIEW* as well. While adding executability of these three features may seem like small progress, they are characteristic (in our experience) of the truly implicit specifications many specifiers, regardless of their expertise and mathematical sophistication, tend to write. Thus, the executability of these features is of key importance in executing specifications. We want executability to be as small a burden on the specifier as possible. Our main goal is to allow the specifier to write purely implicit specifications in a language that is in general not executable, and then execute as much of the specification as is practical. If more executability is needed, then the specification can be refined to an executable form. Thus, specification languages that are designed to be executable are in general not suitable for our purposes, but we do want to ease the trade-off between implicitness and executability as much as we can.

In addition, our technique has been implemented as an interpreter for model-based specifications, and so avoids the problems associated with validating a specification when translation to some programming language is used for execution. The technique is described in the context of the specification language SPECS-C++, which is a VDM-like language specialized for specifying C++ [ES90, Str91] classes. However, our execution technique is not tied to any special feature of SPECS-C++, and so would work as well for VDM specifications as it does for those written in SPECS-C++.

An informal description of SPECS-C++ is presented in the next section. Our approach to executing SPECS-C++ follows in Section 3. Section 4 presents an informal characterization of the limitations of our execution technique. Finally, we examine some of the most closely related work in Section 5 and speculate about the continued evolution of directly executable, abstract, and formal specifications for C++ classes in the concluding Section 6.

2 Syntax for Executable SPECS-C++ Specifications

The next two sections summarize our work on direct execution of model-based specifications. We have a literate[2] Standard ML (SML) [Pau91] implementation of an interpreter for a subset of SPECS-C++. This SML implementation should be thought of as a prototype and a demonstration of the feasibility of the execution technique, as it uses an abstract syntax for SPECS-C++ that is too cumbersome for actually writing specifications. Other members of the SPECS-C++ development team are currently working on tools that use the concrete syntax of SPECS-C++ presented in this section. The next section describes the executable subset of SPECS-C++ and the execution technique used in the SML implementation. In the interest of brevity, no SML code is presented, but is available in [WBL94].

[2]The interpreter is being developed with the aid of the literate programming tool Noweb [Ram91]. Noweb allows a program and the text describing it to be written simultaneously and in the same file, provides a structured way of presenting the code, and makes extracting the code or the descriptive text easy.

A SPECS-C++ specification consists of a set of class specifications and the definitions of the abstract model types used in those specifications. The primitive abstract types are basically those of VDM: `integer`, `float`, `char`, `string`, and `bool`, where `float` is the type of real numbers. The `void` type of C++ is included as the return type of pure procedures. More complex types are composed from these basic types: finite sets, sequences, tuples, and alternative (union) types. The map and function types of VDM are not included, but finite maps and functions may be modeled as sets of tuples. All abstract type definitions are visible globally.

Class specifications in SPECS-C++ roughly correspond to modules in VDM. Each class specification defines a type (with the same name as the class) and a set of operations on that type, called public member functions. These public member functions correspond to the exported operations of a VDM module, and are specified using pre- and post-conditions, much like implicit VDM specifications. However, the precise meaning of a class specification is different than the meaning of a VDM module, because SPECS-C++ is an *interface* specification language [GHW85, GHG$^+$93, Lam89] for C++. That is, a SPECS-C++ specification of a class can only be implemented by a C++ class with the same name, each prototype of a public member function in the specification must appear (with the same name, return type, and arguments) in the implementation as a public member function, and the implementation of each public member function must satisfy the specification given for it in the SPECS-C++ class. For example, consider a SPECS-C++ class `OrderedPair` (representing ordered pairs of integers) and a member function `First` that returns the first element of an `OrderedPair`. (This specification will be presented in full shortly.) The relationship between the specification and the implementation is pictured in Figure 1. The interface and the meaning are both part of the specification that must be matched by the implementation.

As in Larch/Ada [GMP92] and LM3 [Jon92], SPECS-C++ specifications are embedded in implementation code. A SPECS-C++ specification is placed in specially formatted comments within a C++ header file. This automatically makes the interfaces match, and avoids redundancy. Thus, the declaration of the class and the prototypes of the member functions in the specification must be compilable C++ code, and the other parts of the specification (i.e., abstract type declarations, pre- and post-conditions, and so on) are written inside C-style (/* to */) comments, and this style of comment cannot be used otherwise. This separates the C++ clients of the specified class from the C++ code that implements it, so that the implementation can be separately compiled. As the specification is included in the header, the header contains all the information that C++ clients of the class need to know to use the class correctly.

As an example of a SPECS-C++ specification, consider the specification of the classes `OrderedPair` (Figure 2) and `Relation` (Figure 3), where `Relation` is modeled as a set of `OrderedPair`s. Member functions with the class name are constructors, and are typically invoked in client code when class instance declarations are executed. Class `Int_Set`, the return type of `RelTo`, is not shown here as it is uninteresting, but would have to be included for executing this example. The pre-condition of the function `RelTo` specifies that the parameter `key` must be in the domain of the relation, and the post-condition specifies that the result consists of all the integers related to `key` in the relation. The obvious VDM analog of `RelTo`'s post-condition exhibits two of the kinds of assertions that are not executable under the VDM-based techniques discussed earlier, as references to post-state values appear inside quantified assertions, and the value of `result'` is specified using only `\in` (set membership). However, this post-condition is executable using our approach. As this post-condition is also a non-trivial example of the expressive power of these features, the specification of `RelTo` will be used as a running example in the section on executing assertions (Section 3). A more detailed description of SPECS-C++ is presented after this example.

In SPECS-C++, a class specification consists of the abstract data members, an invariant, a set of abstract function definitions, and a set of public member function specifications. Each of these

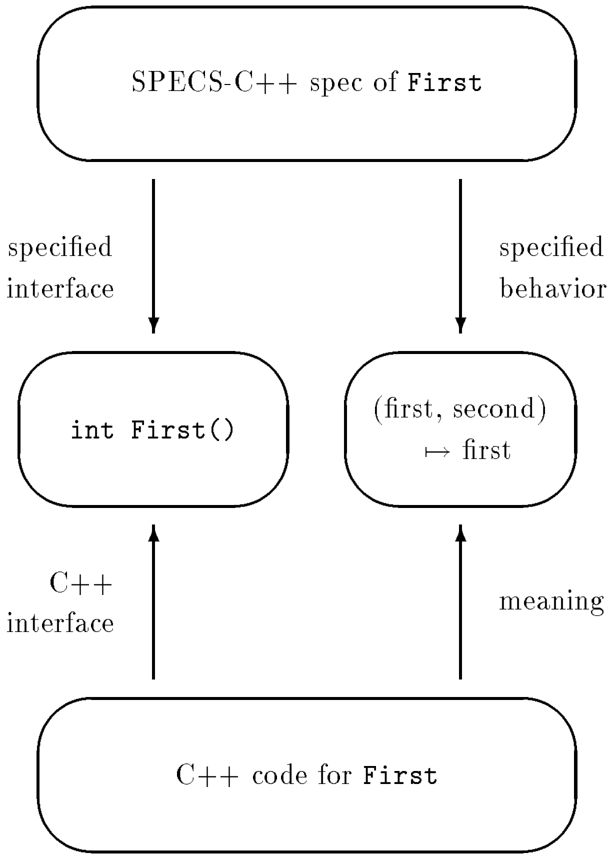

Figure 1: The meaning of a SPECS-C++ member function specification. The C++ code must match the specified interface and have a compatible meaning.

parts is described below.

In C++, the data members of a class implement the state of instances of the class – i.e., elements of the type defined by the class. Abstract data members in SPECS-C++ differ from the concrete data members of C++ in two important ways:

- abstract data members are used to model the state of instances of the class type (and so appear only in the specification), while C++ data members actually implement these states. Hence, many different collections of concrete C++ data members can implement the specified abstract data members.
- abstract data members are visible globally in the specification, while C++ allows various kinds of control over what parts of a program may access concrete data members. The development method used with SPECS-C++ requires that the C++ data members implementing any given abstract data members be private (visible only within the class where they are defined). Thus, class implementations that satisfy a SPECS-C++ specification have only private data members, and all client code access must be through the specified public member functions.

```
class OrderedPair {
/* model
**   data members
**     int first
**     int second
** operations
*/
public:

OrderedPair(int f, int s);
/* modifies: self
** postA: first' = f /\ second' = s
*/

int First();
/* postA: result' = first
*/

int Second();
/* postA: result' = second
*/
};
```

Figure 2: Specification of class `OrderedPair`.

In our semantics for SPECS-C++, class instances are modeled as tuples composed of the abstract data members, and so are similar to VDM composite types (trees). If a class has no abstract data members, then its type is modeled as the empty tuple. So, for example, instances of class `OrderedPair` would be modeled as follows:

```
tuple OrderedPair (int first, int second)
```

The invariant in SPECS-C++ is a first order predicate calculus assertion that every class instance must satisfy. Invariants may also be written for any specifier-defined abstract type. In our example specifications, any pair of integers is a valid `OrderedPair`, and any set of `OrderedPairs` is a valid `Relation`, and so their invariants are both `true`, and hence omitted. The invariant for a class should refer only to the abstract data members of the class it belongs to.

Abstract functions allow a first order assertion on the abstract model to be abstracted and parameterized, much like the functions of an ordinary programming language. Abstract functions cannot be invoked in client code. They are just aids in modularizing other assertions in a specification.

The specification of a member function consists of a C++ function prototype, a pre-condition, a `modifies` clause, and a post-condition. The function prototype includes a return type, the name of the function, and a parameter list with the types and names of the formal parameters, exactly as in C++. Member functions always take a formal parameter of the associated class type which is not explicitly listed, the *default parameter*. In SPECS-C++, this default parameter is always referred to as `self`, and is analogous to `*this` in C++. As a shorthand, the abstract data members of the default parameter can be referred to without mentioning `self`, so `first` and `self.first` are synonyms when specifying the member functions of class `OrderedPair`.

```
class Relation {
/* model
**   data members
**     set of OrderedPair theRel
** operations
*/
public:

Relation();
/* modifies: self
** postA: theRel' = {}
*/

Insert(OrderedPair elem);
/* modifies: self
** postA: theRel' = theRel \union {elem}
*/

...

Int_Set RelTo(int key);
/* preA: \exists (OrderedPair p) [
**         (p \in theRel) /\ p.First() = key]
** postA: \forall (OrderedPair p) [
**          (p \in theRel) /\ p.First() = key => p.Second() \in result'
**        ] /\
**        \forall (int i) [
**          (i \in result') =>
**            \exists (OrderedPair p) [
**              (p \in theRel) /\ i = p.Second() /\ key = p.First()]]
*/
}
```

Figure 3: Specification of class `Relation`.

The `modifies` clause is a list of the formal parameters, possibly including the default parameter, that the member function is allowed to mutate. The `modifies` clause could also be used to advertise mutation of non-local objects, but this is not currently allowed by the SPECS-C++ method. An omitted `modifies` clause means that the member function has no side effects.

Just as in VDM, the pre- and post-conditions describe respectively what must be true for the member function to execute correctly, and what will be true after executing the member function. An omitted pre-condition is equivalent to just `true`, and means that successful execution of the member function does not depend on the pre-state. We take the total correctness approach, so an implementation is required to terminate in a state that satisfies the post-condition whenever the pre-state satisfies the pre-condition. In the post-condition, the keyword `result'` is used to refer to the return value of the member function. To distinguish between the pre-state and post-state values of formal parameters in the post-condition, variable identifiers representing post-state values are primed ('). We include `result'` in the set of primed identifiers. Primed identifiers may not appear in a pre-condition.

Pre- and post-conditions, abstract functions, and invariants are first order predicate calculus assertions written over SPECS-C++ types. Thus, SPECS-C++ expressions include the literals of the primitive abstract types, the normal mathematical operations on integers and floats, and the standard constructors and observers for sets, sequences, and tuples. Equality is defined in the natural way for each of these types. Sets are constructed by the standard `{...}` notation, including set comprehensions, i.e. `{i + 2 | 1 <= i <= 10}`. They are observed by `\in` and `\subset`, which are just membership and subset, respectively. Sequences are constructed by `<...>`, and `||` is used for appending of sequences. Given this much notation, the observers that decompose sequences are defined as follows, where `s` represents a sequence, and `e` is a sequence element:

```
first(<e>||s) = e
last(s||<e>) = e
header(s||<e>) = s
trailer(<e>||s) = s
index(<e>||s, i) = if i = 1 then e else index(s, i - 1)
```

Note that none of these observers may be applied to empty sequences. The only built-in way to observe tuples is to look at their components, or fields. For each field of every tuple type, SPECS-C++ provides an observer with the same name as the field that returns the associated value. For example, given a declaration:

```
tuple rational (int num, int denom)
```

functions `num: rational -> int` and `denom: rational -> int` are then available for extracting the components of the tuple.

Predicate calculus assertions are built with the standard boolean connectives: logical and (`/\`), or (`\/`), implication (`=>`), and negation (`!`), and the universal (`\forall`) and existential (`\exists`) quantifiers.

Abstract function references and calls of member functions with non-void return types are also allowed in assertions. Any side effects of the called member function are ignored, so the meaning is that the value[3] defined by the body of the abstract function, or respectively the value of `result'`

[3]While the specifications of member functions can in general be nondeterministic, we require abstract functions and member functions that are referenced in pre- or post-conditions to be deterministic. Otherwise, the semantics of assertions becomes problematic. For example, consider the truth value of the assertion `foo(i) = 3` when `foo(i)` can evaluate to either 2 or 3.

defined by the post-condition of the member function, is substituted for the call or reference in the assertion. (A syntax and informal semantics for using side effects of member functions in assertions has been developed, but this feature is yet to be formalized.)

3 An Algorithm for Executing Assertions

Only public member functions of SPECS-C++ classes can be referenced in client code, so the direct execution of SPECS-C++ (from a client's perspective) is just the execution of public member function calls.[4] A call to `RelTo` would have the form `R.RelTo(2)`, where `R` is an instance of class `Relation`. Note that the parameters are not C++ values, but rather abstract values corresponding to C++ values. To interpret such a SPECS-C++ public member function call, we use its specification as follows:

1. Check that the actual arguments satisfy the pre-condition of the called member function.
2. Construct post-state values that satisfy the post-condition of the called member function.

If the post-condition isn't satisfiable, or if the execution technique isn't adequate for the given post-condition, then this attempt will fail.

3.1 Evaluating Pre-conditions

As the pre-condition is evaluated in the pre-state of the operation, all the values in the pre-condition assertion are known – they are exactly the actual arguments of the member function. Thus, all that is needed is to apply the definitions of the built-in operators, and the implementation is straightforward. The quantified assertions and set comprehension expressions are the only possible complications.

In the executable subset of SPECS-C++, universally quantified assertions must be of the form:

```
\forall (T x) [ (BP(x)) /\ P => Q ]
```

and existentially quantified assertions must be of the form:

```
\exists (T x) [ (BP(x)) /\ P ]
```

where `T` is the type of the bound variable, and `BP(x)` is either `x \in E` and `E` is a finite set or sequence, or `BP(x)` is `low <= x <= high`, for some integer valued expressions `low` and `high`. We use the term *domain* of the quantified variable to mean either the predicate `BP(x)` or the set of values that satisfy that predicate. Which meaning is intended should be clear from context. Clearly, quantification is restricted to finite domains in the executable subset. The assertions `P` and `Q` are arbitrary assertions of the executable subset (including quantified assertions), and the `/\ P` portion of both quantifier forms is optional.

A universally quantified assertion such as:

```
\forall (int x) [(1 <= x <= 5) => x < 6 ]
```

is evaluated by applying the predicate `x < 6` to each of the integers 1 through 5, and logically "anding" together each of the results. Thus, evaluating universal quantification corresponds to an *and reduction* over the domain of the quantified variable. For existential quantifiers, an assertion such as the pre-condition of `RelTo`:

[4]Invariants could also be executed in order to check that all class instances created or modified by member functions satisfy their respective invariants. However, we have not done so to-date.

```
\exists (OrderedPair p) [(p \in theRel) /\ p.First() = key ]
```

is evaluated by applying the predicate `p.First() = key` to each `OrderedPair` in `theRel`, and logically "oring" together all of the results – which corresponds to an *or reduction*. This technique of evaluating quantifiers with reductions is common in the literature [BM93].

In the executable subset, set comprehensions must be of the form:

```
{F(x) | (BP(x)) /\ P}
```

where `BP(x)` and `P` are used in precisely the same manner as they were in the definitions of the executable quantified assertions. The term `F(x)` is an arbitrary term of the executable subset of SPECS-C++. The term `F(x)` is not required to contain any occurrence of `x`, but will in any interesting case. Given this definition, the evaluation of a set comprehension is basically a filter of the domain, followed by a map. For example, the expression:

```
{x + 2 |(1 <= x <= 5) /\ x mod 2 = 0}
```

is evaluated by choosing the integers between 1 and 5 that are even, adding 2 to each, and making a set out of the results.

3.2 Satisfying Post-conditions

Executing a post-condition is more involved. The following steps provide an overview of our algorithm for executing a post-condition:

1. Split the post-condition into constructive and nonconstructive parts. The constructive parts are those which will actually be helpful in constructing post-state values that satisfy the post-condition.

2. Generate a list of constraints from the constructive part. Each constraint defines a post-state value or some part of a post-state value.

3. Solve the constraints to construct the portion of the post-state that differs from the pre-state. The constraints need not determine unique post-state values (and so some "looseness" in executable specifications is possible), but this process is deterministic. Thus, calling the same sequence of member functions with the same arguments will always produce the same results.

4. Check the nonconstructive portion of the post-condition by evaluating it in a state where unprimed identifiers have their pre-state values, and primed identifiers (including `result'`) have their post-state values. This state is built from the pre-state and the state constructed in the previous step. If this check fails, then the execution of the post-condition fails.

5. Construct the state that results from the member function call. This state reflects mutations of the actual parameters of the call.

We are planning to generalize our algorithm by supporting backtracking; if the evaluation fails in step three or step four, the algorithm could go back to step one and make a different split into constructive and non-constructive parts, or to step three and try some of the other possibilities that any looseness in the specification permits. We will discuss the possibilities for backtracking in more detail as the algorithm unfolds.

3.2.1 Splitting into Constructive and Nonconstructive Parts

The first step is to split the post-condition into two parts: the part that will be useful in constructing post-state values, and the part that can only be used as a check on post-state values. This splitting is based on the structure of the post-condition. For example, in an assertion of the form `A1 /\ A2`, both `A1` and `A2` must be true for the post-condition to be satisfied, and so each is processed separately.

With an assertion of the form `A1 \/ A2`, the splitting algorithm tries to determine if either `A1` or `A2` is necessarily false, just using pre-state values. If either is necessarily false, it discards that argument and continues processing the other. For example, in the post-condition: `(x = 3 /\ x' = 4) \/ (x != 3 /\ x' = 3)` (recall that primed variable identifiers represent post-state values), one argument of the `\/` assertion will be false just from pre-state values, and so can be ignored. If neither argument can be determined to be false, then the entire `\/` assertion is nonconstructive, and is evaluated only after the construction of all post-state values. For example, the entire assertion: `x' = 3 \/ x' = 4` is nonconstructive. However, when backtracking is added to the interpreter, this case can be handled differently. The interpreter can assume that `x' = 3` is true, and proceed with the execution. If this assumption results in an unsatisfiable or non-executable post-condition, then the interpreter can backtrack and make the other possible assumption (`x' = 4`). If this assumption fails, then the post-condition must be unsatisfiable or non-executable.

The first step in evaluating an assertion of the form `A1 => A2` is to determine whether the antecedent `A1` is necessarily true or false, in the same way that the arguments of an `\/` assertion are tested. If `A1` must be false, then the entire `=>` assertion is ignored, as there is no requirement that the consequent `A2` holds. If `A1` must be true, then `A2` must hold, and so is processed recursively. If the truth value of the antecedent cannot be determined just from the pre-state values, then the entire assertion is nonconstructive. Again, the addition of backtracking allows better handling of this case. The two possible assumptions that can be made are that `A1` and `A2` are both true, or that `A1` is false.

Of the post-condition operators that are not logical connectives, the most important are `=`, `\in`, and `\subset`. These are the ones that directly contribute to the construction of post-state values. For one of these operators to be constructive, it must be the case that one argument of `=` and the left argument of both `\in` and `\subset` contain no primed identifiers. This allows the value of that argument to be computed in the pre-state in much the same way that pre-conditions are evaluated. The other argument of `=`, and the right argument of both `\in` and `\subset`, must represent a post-state value or part of a post-state value. In other words, it needs to be a primed identifier, or arbitrarily many applications of the built-in tuple and/or sequence observers to a single primed identifier. (The tuple and sequence observers were discussed in Section 2.) For example, `x' = 3`, `elem \in theRel'`, and `{3} \subset result'` are all constructive, but `result' = self'.Second()`, `5 * x' = 3`, `x' in {3, 4}` and `index(S', i') = 3` are not. Clearly, "constructive" information can be obtained from assertions that are not currently constructive, and so this is an area of ongoing research. In Section 4, we discuss an iterative technique for relaxing these restrictions.

Other kinds of relational operators and negated assertions provide some information about post-state values, but are usually of little help in actually constructing them, and so are classified as nonconstructive. For example, `x' < 3` or `!(x' = 3)` do give some information about the post-state value of `x`, but both are satisfied by an infinite number of post-state values. One possible extension of the work presented here is to use constraint-satisfaction techniques [Lel88] both for gleaning more information from these kinds of expressions and for generalizing what can represent a post-state value in a constraint.

Set comprehensions, abstract function references, and calls to member functions are constructive if they contain no occurrences of primed identifiers. For a universally quantified assertion to be constructive, the expression for the (finite) domain that the quantified variable ranges over and the assertion `P` (the rest of the antecedent of the implication) must not contain primed identifiers. Additionally, the assertion `Q` (the consequent of the implication) must be constructive, using the definition being developed in this section. Otherwise, the universally quantified assertion is nonconstructive. For example, in the post-condition of `RelTo`:

```
\forall (OrderedPair p) [
  (p \in theRel) /\ p.First() = key => p.Second() \in result']
```

is constructive, as neither the domain of the quantified variable nor the antecedent of the implication contains a primed identifier, and the consequent of the implication is constructive. However,

```
\forall (int i) [
  (i \in result') =>
    \exists (OrderedPair p) [
      (p \in theRel) /\ i = p.Second() /\ key = p.First()]]
```

is not constructive, because `result'` is the domain of the quantified variable.

Existentially quantified assertions are constructive if the domain of the bound variable contains no primed identifiers, and the rest of the assertion is constructive.

3.2.2 Evaluating Constructive Parts into Constraints

To simplify the construction of post-state values, the constructive part of the post-condition is transformed into a list of constraints on the output values. As one might expect from the last section, these constraints are equality, membership, and subset, and any occurrence of one of these operators is immediately evaluated into a constraint, with the argument that contains no references to post-state values evaluated into a value of one of the abstract types. Note that this transformation is only applied to constructive expressions as defined in the previous section, so an appropriate argument with no references to post-state values is guaranteed to exist. For the same reason, the transformation need deal only with `/\` and quantified assertions, and an `/\` assertion is handled by simply appending the lists of constraints generated by its two arguments. For example, given the the post-condition `header(s') = <1, 2> /\ last(s') = 3`, the transformation produces the constraint list `[header(s') = <1, 2>, last(s') = 3]`.

Universally quantified assertions are transformed to constraints by repeatedly evaluating the body of the assertion with the variable bound by the quantifier bound to each value in the domain, in turn. The resulting lists of constraints are appended into a single list of constraints. So, if (in the post-condition of `RelTo`) the default parameter to `RelTo` is `{(1, 2), (2, 2), (2, 3)}` and the value of `key` is `2`, then the assertion

```
p.First() = key => p.Second() \in result'
```

is evaluated 3 times, with `p` bound to `(1, 2)`, `(2, 2)`, and `(2, 3)` in turn. The resulting constraint list is `[2 \in result', 3 \in result']`.

For existentially quantified assertions, the technique is to repeatedly evaluate the body of the assertion for its boolean value, where subexpressions involving post-state values are ignored. This is done with the quantified variable bound to each element of the domain, in turn, until the evaluation returns true. Then, the element that satisfies the body is bound to the quantified variable, and the body is evaluated to generate constraints. So, for example, an assertion such as:

```
\exists (int x) [(1 <= x <= 3) /\ x mod 2 = 0 /\ y' = x + 5 ]
```

causes the predicate `x mod 2 = 0` to be evaluated for `x` equal to 1, 2, and 3. As only 2 satisfies this predicate, the constraint list `[y' = 7]` is generated. If multiple elements of the domain satisfy the predicate, the first one that this evaluation technique finds is used. Note that such an existentially quantified assertion constitutes an underdetermined or non-deterministic specification, and that this is the only kind of "loose" specification that the execution technique can use constructively in its current form. With the addition of backtracking, specifications that are loose because of assertions using `\/` and `=>` as described earlier can also be used constructively. The addition of backtracking will also allow more sophisticated use of existentially quantified assertions. Many elements of the domain quantified over can potentially satisfy the parts of the body of the assertion that do not refer to post-state values. In case the original choice of such an element causes the execution of the post-condition to fail, backtracking would allow other elements of the domain to be tried.

3.2.3 Evaluating Constraints into Values

The execution algorithm next tries to construct a post-state value for identifiers appearing in the `modifies` clause and for any non-void function result, using the list of constraints constructed as described in the previous section. The first step is to check whether the (primed) identifier appears directly in one or more constraints. If so, then there are only two possibilities, assuming that the post-condition used to generate the constraints is satisfiable. Either one or more identical `=` constraints match, and the (identical) post-state value is directly in each such constraint, or a group of `\in` and `\subset` constraints match, and so a value of type set needs to be constructed. This is done by unioning together the first arguments of all the constraints that matched. For example, if the list of constraints is `[2 \in result', 3 \in result']`, then the post-state value constructed for `result'` is `{2, 3}`.

If the primed identifier doesn't match any constraint and the post-state value under construction is of type tuple or sequence, then the next step is to check the constraint list for applications of built-in tuple or sequence observers, respectively. (Recall that these observers are defined in Section 2.) The only non-atomic values in SPECS-C++ are tuples, sequences, and sets, and the only built-in observers of sets (`\in` and `\subset`) are converted directly into constraints, so tuples and sequences are the only types whose values can be constructed in this way.

The only built-in way to observe tuples is to look at their components, or fields. Thus, the next step in trying to construct a post-state value of a tuple type is to attempt to match (in the constraints list) the application `field(identifier')` for each field of that type of tuple. If this is successful, then the post-state tuple value can easily be constructed.

When trying to construct a post-state value of a sequence type, our algorithm constructs applications of the sequence observers to the appropriate primed identifier, and then matches these applications with the constraint list. Each successful match defines a part of the post-state value. For example, if the primed variable of interest is `s'` and the constraint list is
`[header(s') = <1, 2>, last(s') = 3]`, then two applications match, and the post-state value of `<1, 2, 3>` can easily be constructed. Clearly, such multiple matches can produce redundant information about the post-state value of a sequence, and this information should be checked for consistency. This has not yet been implemented, but seems straightforward. The other case (some part of the post-state value of a sequence is not defined in the constraint list) is treated next.

If some part of the post-state value of a sequence or tuple is completely unconstrained by the constraint list, then there are two possibilities, depending on whether the associated identifier (formal parameter of a tuple or sequence type) is defined (has a value associated with it) in the pre-state. If the identifier is defined in the pre-state, then the "and nothing else changes" semantics of SPECS-C++ implies that the unspecified part should be whatever it was in the pre-state value. If the value is being constructed from scratch, then the post-condition didn't provide enough information to construct it, and so the construction fails and reports an error.

Post-state values of union types are constructed by searching the types of the union, and finding the first one to which the value belongs. If none of the type works, then the attempt to construct the post-state value fails.

The steps described in the previous paragraphs are applied in a mutually recursive fashion. Thus, if we had modeled integer relations as a sequences of sets of integers (i.e. each index position in the sequence is related to each of the elements of the set found at that index position), then our algorithm could utilize constraints like `3 \in index(R', key)` in constructing a post-state value for `R'`.

If the post-state value of some identifier appearing in the `modifies` clause is completely unspecified (i.e. that identifier does not appear in any constraint), then it is handled in the same way as unspecified parts of tuples or sequences – if it is defined in the pre-state, then the post-state value is the same as that of the pre-state value, and if it is not defined in the pre-state, then the construction fails. Note that the `result'` of the member function being specified is always undefined in the pre-state, and so member functions with non-void return types must always construct a post-state value for `result'`.

3.2.4 Checking the Nonconstructive Parts

Next, the parts of the post-condition that were determined to be nonconstructive are checked by evaluating them for their boolean values. This evaluation is done with respect to *both* the pre- and post-states, as the truth value of the nonconstructive parts may depend on both. Unprimed identifiers have their pre-state values, and primed identifiers (including `result'`) have their post-state values. So, for example, the nonconstructive portion of the post-condition for `RelTo`:

```
\forall (int i) [
  (i \in result') =>
    \exists (OrderedPair p) [
      (p \in theRel) /\ i = p.Second() /\ key = p.First()]]
```

is evaluated w.r.t a pre-state where `theRel` is `{(1, 2), (2, 2), (2, 3)}`, `key` is 2, and a post-state where `result'` is `{2, 3}`. The evaluation mechanism is identical to that used for pre-conditions. If this evaluation returns true (as it does for this example), then all is well. Otherwise, the attempt to execute the post-condition fails. This can occur because the post-condition isn't satisfiable, or because the technique described for finding post-state values failed to find satisfactory ones.

3.2.5 Constructing the Result State

To complete the execution of a member function call, we need to construct the state that results from the call. This is different from the post-state we have been discussing so far, as this is outside the scope of the member function. Thus, the formal parameters are no longer part of the state, and any side effects of the member function are now reflected in the actual parameters, rather than the formals. Note that any formal that is mutated must appear in the `modifies` clause of the called function. If the member function that was called has a non-void return type, then the value constructed for `result'` is just printed, as return values of functions do not affect the state.

As all state modifications performed by the member function have been made and any return value constructed, execution of the member function call is complete.

4 Limitations of the Execution Technique

While the work described here makes a large subset of SPECS-C++ executable, not all valid SPECS-C++ specifications can be executed. A post-condition must contain constructive subassertions that define all of the post-state values to be found. The following list of nonconstructive (and so, non-executable) assertions is provided to highlight the limitations of the execution technique.

- Quantified assertions where the domain of the bound variable depends on post-state values, implications where the antecedent refers to post-state values, and multiple occurrences of primed identifiers in the same constraint (in the sense of Section 3.2.3). Examples include:

  ```
  \forall (int x) [(x \in (foo' \union {3})) => x \in foo2' ]

  x' = 3 => y' = 4

  index(S', i') = 3
  ```

  The backtracking approach previously described in only a partial solution to these problems. Many more such assertions would be executable by a more incremental approach: find post-state values for all the identifiers possible, and then try again in a state where these primed identifiers are bound to the value found. This corresponds to finding all of the post-state values that depend only on pre-state values, then finding all the post-state values that depend only on pre-state values and those post-state values found in the previous step, and so on until no more post-state values can be found. Note that this scheme is guaranteed to terminate, as at least one post-state value must be found at each step for this iteration to continue.

- Negated assertions and assertions in which any relational operator other than `=`, `\in`, or `\subset` is used to describe the relationship of some value to the post-state value being constructed. Examples of such assertions can be found in Section 3.2.1. They limit the possible post-state values that can be constructed. As these limits are checked when the nonconstructive portion of the post-condition is evaluated, the evaluation technique already utilizes such assertions as well as can be expected in the absence of constraint-satisfaction techniques such as those discussed in [Lel88].

- Post-state values used as arguments to abstract function references and member functions calls in post-conditions. Assertions like `someabsfun(foo')` do not help in constructing the post-state value for `foo`, at least as far as the work described here is concerned. Note that simply replacing the call or reference by the specification of the associated function will not work. To see this, consider a post-condition of some member function of class `Relation` that just mimics `RelTo` on a `key` of 2, i.e. `result' = self.RelTo(2)`. The post-condition of `RelTo` is a (boolean valued) assertion, so this new post-condition fails to type check. The same problem arises with abstract function references, as an abstract functions can also define its return value in the implicit manner typified by `RelTo`. However, this problem does seem reasonably tractable, and so this is an area for further research.

5 Related Work

The most closely related work is the **fase3/C++** language of Kamin and Kraus [KK93, Kra88], as it is a (mostly) executable interface specification language for C++ classes. The **fase3** approach is very similar to that taken in the Larch [GHW85, GHG$^+$93] family of specification languages, as it is two-tiered, consisting of the **fase3** shared language and the bf fase3/C++ interface language for C++. The shared language is where most specification occurs, and so where interesting execution occurs as well. Only a few primitive types such as integer, boolean, and so on are built into **fase3**. More structured types, such as the set, sequence, and tuple types of SPECS-C++ are specified in the shared language using a unique style of algebraic specification that allows any type to be represented as a tuple of functions. As long as these functions remain finite for a particular element of the type, the functions can be represented as "tables" (finite sets of tuples), and quantified assertions over that element can be executed. The syntax for quantified assertions is elegant and concise, as the specifier need not supply an explicit bound on the quantified variable, as is required in executable SPECS-C++ specifications. Assertions that use observers to define values are also executable — in fact, the functions that represent an element are exactly the observers of the type as applied to that element.

However, many quantified assertions that are executable using our technique for SPECS-C++ can not be executed in **fase3**. Given the natural **fase3** shared language specifications of set, sequence, and tuple, the kinds of quantified assertions that have no references to post-state[5] values that are executable seem quite similar to those that are executable using our technique for SPECS-C++. However, only one kind of **fase3** quantified assertion can be evaluated if it contains references to post-state values, and so for more than just its boolean value. This kind of assertion is a restricted form of existential quantification that can only be used to select a particular element from the domain quantified over. Thus, for example, the post-condition of the member function `RelTo` as discussed in the previous section, is not executable using the **fase3** execution technique.

The execution techniques are quite different. In **fase3**, specifications are first compiled to an extended λ-calculus, and then to an extended form of combinator graph, which is then reduced. An execution technique of this generality seems to be necessary because the specifier defines the observers of a particular type, rather than the observers being built into the language as in SPECS-C++. Because of the compilation phases and complicated reduction phase, the **fase3** execution technique seems unlikely to execute specifications as rapidly as our technique for SPECS-C++. We are also unsure of the usefulness of errors reported by the reduction algorithm in debugging the specification that the graph was derived from.

Another work with some relation to our own is the structural mapping from Object-Z [CDD$^+$90] to C++ of Rafsanjani et. al. [RC93]. This mapping is an informal guideline for producing C++ implementations from Object-Z specifications. Object-Z classes are mapped to C++ classes, operations to virtual member functions, and so on. The mapping is not intended to be an automated translation, so the only tool support provided is a partial implementation of the Object-Z basic types. The work does provide some interesting observations on the relationship between specification inheritance and code inheritance which may be helpful as we work on adding inheritance to our interpreter for SPECS-C++.

[5]In **fase3**, there is no concept of pre- and post-state at the shared level. When we refer to post-state values in the context of **fase3**, we are referring to the values returned from shared language functions. These are the values that are defined by specifications, and so play the role of post-state values in SPECS-C++.

6 Conclusion

One of the strengths of the execution technique described here is that it is relatively efficient (at least with respect to Prolog) and execution times are predictable. The most expensive built-in operators (set intersection, difference, and subset) take $O(n^2)$ time, execution of quantifiers takes time linear in the size of the domain quantified over, and all other built in operations take either linear time (length of sequences, size of sets, etc.) or constant time. Hence, running times can be estimated by inspection of the specification. With the addition of backtracking to the interpreter, some of this predictability would be lost. However, the loss of efficiency would be minimal, as any post-condition that is currently executable would not require backtracking for correct execution by the augmented interpreter.

However, this execution technique does lose some aspects of expressiveness as compared to Prolog, even with the addition of backtracking. For example, a classic way of implementing sorting in Prolog is to define predicates "permutation" and "ordered", and then define a sorted sequence as an ordered permutation of the original. SPECS-C++ is expressive enough to state the specification of a sort this way – the post-condition would look like:

```
Permutation(s, s') /\ Ordered(s')
```

for a sequence argument `s` and appropriate abstract functions `Permutation` and `Ordered`. This specification is not executable by the technique described here. On the other hand, executing the Prolog sorting program takes exponential time, and so there is a trade-off between expressiveness and execution time.

Additionally, our execution technique was designed to take advantage of the ways in which humans usually write specifications. We have found, for example, that people tend to write more constructive specifications (in the sense developed in Section 3.2.1), rather than nonconstructive specifications like the one discussed in the previous paragraph. In particular, our technique was developed in the context of a suite of specification examples used for teaching formal methods at both the undergraduate and graduate levels. Almost all of these specifications are executable with only (very) minor modifications, such as placing parenthesis around the domains of variables bound by quantifiers. This provides good evidence that the executable subset of SPECS-C++ is useful in practice.

We have provided an interpreter for SPECS-C++, rather than using translation to some programming language. Hence, this technique does not require the user to know Standard ML (the language the interpreter is written in), and also can more easily report helpful error messages than techniques using translation.

The execution technique also demonstrates the use of default frame axioms in a specification language. Frame axioms are used to say "and nothing else changes" – that only the state transformations explicitly required by the post-condition actually occur, and no more. The modifies clause is a form of frame axiom, as it explicitly limits the side effects of a member function to only the formals (and corresponding actuals) that occur in the modifies clause. However, additional frame axioms are often required, as member function specifications don't always completely specify all post-state values. While extra explicit frame axioms can be included in the post-condition, doing so leads to large and unreadable specifications, along with a number of other problems [BMR93]. The technique used in executing SPECS-C++ is to embed default frame axioms in the specification language. This is demonstrated in the section on evaluating constraints into post-state values (Section 3.2.3). For sets, the default frame axiom is to find a minimal set satisfying the post-condition. To see this, note that nothing is included in a post-state set value unless the post-condition explicitly includes it, in the form of a `\in` or `\subset` expression. For tuples and sequences, the default frame axiom is that no field or index position changes from the pre-state to the post-state unless the post-condition explicitly specifies that it change.

Two major extensions of the interpreter are planned. These extensions will add constructs for dealing with SPECS-C++ abstractions of C++ objects and for specification inheritance.

In C++, "An *object* is a region of storage" [Str91]. Equivalently, an object may be thought of as a cell in memory, or a value and its address. In SPECS-C++, we are concerned with distinguishing between objects and values because objects can be mutated and aliased, while values cannot. Incorrect usage of mutation and aliasing is a common cause of errors in C++ programs, so it is important to advertise to users of a class (through the specification of the class) exactly where mutation and aliasing can occur. SPECS-C++ borrows the C++ notion of references as a uniform mechanism for creating and handling objects. However, references are not yet part of the executable subset of SPECS-C++.

Just as in C++, in SPECS-C++ inheritance relationships are established through the use of derived classes. The syntax and semantics used are similar to C++, so that a class specified as a derived class inherits the abstract data members and (the specifications of the) member functions of its base (super) class [Lea93]. We plan to add specification inheritance to the SPECS-C++ interpreter, and so to the executable subset of SPECS-C++.

Even in its unfinished state, the SPECS-C++ interpreter can already be used in two important ways. The first is that it provides a formal semantics for a large subset of SPECS-C++. Even if the specifier never wants to execute specifications, this work is useful in that it gives denotational and operational semantics for the executable subset of SPECS-C++, which includes the built-in operators, abstract functions, using member functions in specifications, and (soon) objects and inheritance of specifications. As the SML compiler provides operational semantics for all the constructs in the executable subset, this work has a decided advantage over non-executable descriptions in reference manuals. The SPECS-C++ interpreter is an unambiguous specification of the meaning of the executable subset of SPECS-C++ specifications.

Secondly, this work provides an executable subset of SPECS-C++, and the means to execute specifications written in the subset. Thus, a specification can serve as a prototype of the finished system. Some of the advantages for the specifier and client are [WBL93]:

1. Validating specifications. The specifier can now test and debug a specification in much the same way that a programmer would validate a program. This pushes validation into the specification stage of the software development process.

2. Understanding formal specifications. The client, who is likely to have little or no experience with formal methods, now has a way to understand a formal specification. By experimenting with the prototype, the client can discover and report to the specifier erroneous or unexpected results, and missing or incomplete features.

Both these points imply that requirements and specification errors are likely to be discovered earlier, resulting in less expensive and more reliable software. Additionally, problems that traditional software engineering techniques cope with poorly, such as missing functionality and incomplete requirements documentation, are much more likely to be addressed by an executable specification. Often, the software developer is completely unaware of such problems, and only discovers them after the completed software is delivered to the client. Giving the client a prototype can result in such problems being discovered far earlier.

Thus, this research is a contribution to both the theoretical and practical sides of formal methods. On the theoretical side, it demonstrates an executable specification language and execution technique with clear advantages over other approaches. On the practical side, it demonstrates that prototypes generated by executable specifications can be efficient enough for practical use. As such, it represents solid progress in applying formal methods to industrial software production.

References

[BL90] V. Berzins and Luqi. Languages for Specification, Design, and Prototyping. In P. A. Ng and R. T. Yeh, editors, *Modern Software Engineering: Foundations and Current Perspective*, chapter 4, pages 83 – 118. Van Nostrand Reinhold, New York, 1990.

[BM93] Paulo Borba and Silvio Meira. From VDM Specifications to Functional Prototypes. *The Journal of Systems and Software*, 21(3):267 – 278, June 1993.

[BMR93] Alex Borgida, John Mylopoulos, and Raymond Reiter. ... And Nothing Else Changes: The Frame Problem in Procedure Specifications. In *Proceedings Fifteenth International Conference on Software Engineering*, Baltimore, May 1993.

[CDD$^+$90] D. Carrington, D. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: An Object-Oriented Extension to Z. In S. Vuong, editor, *Formal Description Techniques, II (FORTE'89)*, pages 281–296. Elsevier Science Publishers (North-Holland), 1990.

[DKC90] A.J.J. Dick, P.J. Krause, and J. Cozens. Computer Aided Transformation of Z into Prolog. In J.E. Nicholls, editor, *Z User Workshop, Oxford 1989*, Workshops in Computing, pages 71–85. Springer-Verlag, 1990.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1990.

[GHG$^+$93] John V. Guttag, James J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, 1993.

[GHW85] John V. Guttag, James J. Horning, and Jeannette M. Wing. The Larch Family of Specification Languages. *IEEE Software*, 2(4), September 1985.

[GMP92] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal Verification of Ada Programs. In Ursala Martin and Jeanete M. Wing, editors, *First International Workshop on Larch, Dedham 1992*, pages 104–141. Springer-Verlag, 1992.

[Hay87] I. Hayes, editor. *Specification Case Studies.* International Series in Computer Science. Prentice-Hall, 1987.

[Hen86] Peter Henderson. Functional Programming, Formal Specification, and Rapid Prototyping. *IEEE Transactions on Software Engineering*, SE-12(2), February 1986.

[HI86] Sharam Hekmatpour and Darrel C. Ince. A Formal Specification-Based Prototyping System. In D. Barnes and P. Brown, editors, *Software Engineering 86*, pages 317 – 335. Peter Peregrinus Ltd., London, UK, 1986.

[HI88] Sharam Hekmatpour and Darrel C. Ince. *Software Prototyping, Formal Methods, and VDM.* Addison-Wesley, Wokingham, England, 1988.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[Jon92] K. D. Jones. A Semantics for Larch/Modula-3 Interface Language. In Ursala Martin and Jeanete M. Wing, editors, *First International Workshop on Larch, Dedham 1992*, pages 142–158. Springer-Verlag, 1992.

[KK93] Samuel Kamin and Tim Kraus. Executable Specifications of C++ Classes. submitted for publication, 1993.

[Kra88] Tim Kraus. The FASE3 System for Executable Data Type Specification. Master's thesis, University of Illinois, Urbana, Illinois, 1988. Technical Report 87-1789.

[Lam89] Leslie Lamport. A Simple Approach to Specifying Concurrent Systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[LB89] J. Leszczylowski and J.M. Bieman. PROSPER: A Language for Specification by Prototyping. *Computer Languages*, 14(3):165–180, 1989.

[Lea93] Gary T. Leavens. Inheritance of Interface Specifications (Extended Abstract). Technical Report 93-23, Iowa State University, Department of Computer Science, September 1993. Appears in the Workshop on Interface Definition Languages, WIDL '94. Available by anonymous ftp from ftp.cs.iastate.edu or by e-mail from almanac@cs.iastate.edu.

[Lel88] Wm Leler. *Constraint Programming Languages.* Addison-Wesley, Reading, Massachusetts, 1988.

[LL91] Peter Gorm Larsen and Poul Bøgh Lassen. An Executable Subset of Meta-IV with Loose Specification. In *VDM '91: Formal Software Development Methods.* VDM Europe, Springer-Verlag, March 1991.

[O'N92a] Guy O'Neill. Automatic Translation of VDM Specifications into Standard ML Programs. Technical Report DITC 196/92, National Physical Laboratory, Teddington, Middlesex TW11 OLW, United Kingdom, February 1992.

[O'N92b] Guy O'Neill. Automatic Translation of VDM Specifications into Standard ML Programs. *The Computer Journal*, 35(6):623–624, December 1992.

[Pau91] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, Cambridge, 1991.

[Ram91] Norman Ramsey. Literate Programming Tools Need Not Be Complex. Technical Report CS-TR-351-91, Princeton University, 1991.

[RC93] G-H. B. Rafsanjani and S. J. Colwill. From Object-Z to C^{++}: A Structural Mapping. In J. P. Bowen and J. E. Nicholls, editors, *Z User Workshop, London 1992*, Workshops in Computing, pages 166–179. Springer-Verlag, 1993.

[Spi88] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.

[Spi89] J. M. Spivey. An Introduction to Z and Formal Specifications. *Software Engineering Journal*, January 1989.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, New York, second edition, 1992. ISBN 013983768X.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, second edition, 1991.

[TC89] R.B. Terwilliger and R.H. Campbell. PLEASE: Executable Specifications for Incremental Software Development. *Journal of Systems and Software*, 10(2), September 1989.

[WBL93] Tim Wahls, Albert L. Baker, and Gary T. Leavens. An Executable Semantics for a Formalized Data Flow Diagram Specification Language. Technical Report TR93-27, Department of Computer Science, Iowa State University, Ames, Iowa 50011, November 1993. available by anonymous ftp from ftp.cs.iastate.edu and by e-mail from almanac@cs.iastate.edu.

[WBL94] Tim Wahls, Albert L. Baker, and Gary T. Leavens. The Direct Execution of SPECS-C++: A Model-Based Specification Language for C++ Classes. Technical Report TR94-02, Department of Computer Science, Iowa State University, Ames, Iowa 50011, February 1994. available by anonymous ftp from ftp.cs.iastate.edu and by e-mail from almanac@cs.iastate.edu.

[WE92] M.M. West and B.M. Eaglestone. Software development: two approaches to animation of Z specifications using Prolog. *Software Engineering Journal*, 7(4):264–276, July 1992.

[ZS86] P. Zave and W. Schell. Salient Features of an Executable Specification Language and Its Environment. *IEEE Transactions on Software Engineering*, 12(2):312 – 325, February 1986.

A An Interpreter for SPECS-C++

This appendix contains the complete code for the SPECS-C++ interpreter, written in Standard ML of New Jersey.

```
fun fst(one, two) = one;
fun snd(one, two) = two;

fun null([]) = true
  | null(l::ls) = false;

fun filter([], _) = []
  | filter(e::el, pred) =
      if pred(e) then e::filter(el, pred) else filter(el, pred);

fun andred([], f) = true
  | andred(e::es, f) = f(e) andalso andred(es, f);

fun orred([], f) = false
  | orred(e::es, f) = f(e) orelse orred(es, f);

fun reduce([], f, zero) = zero
  | reduce(e::es, f, zero) = f(e, reduce(es, f, zero));

fun find(l::ls, pred) =
      if pred(l) then l else find(ls, pred);

fun isin([], pat) = false
  | isin(l::ls, pat) = pat = l orelse isin(ls, pat);

fun len l = length l;

signature VALUE =
  sig
  type V
    exception Set_not_homogeneous
    exception Sequence_not_homogeneous
    exception Arg_not_set
    exception Arg_not_sequence
    exception Type_error
    exception Sequence_empty

    val mkset: V -> V
    val emptyset: V
    val union: V * V -> V
    val intersection: V * V -> V
    val difference: V * V -> V
    val subset: V * V -> V
    val size: V -> V

    val mkseq: V -> V
    val emptyseq: V
    val append: V * V -> V
    val first: V -> V
    val header: V -> V
    val last: V -> V
    val trailer: V -> V
    val length: V -> V
    val member: V * V -> V
    val index: V * V -> V

    val mktuple: (string * V) list -> V
    val field: string * V -> V

    val add: V * V -> V
    val sub: V * V -> V
    val divide: V * V -> V
    val mult: V * V -> V
    val modulo: V * V -> V
    val equal: V * V -> V
    val less: V * V -> V
    val lesseq: V * V -> V
    val greater: V * V -> V
    val greatereq: V * V -> V
    val andexp: V * V -> V
    val or: V * V -> V
    val implies: V * V -> V
    val neg: V -> V

    val intvalue: int -> V
    val realvalue: real -> V
    val stringvalue: string -> V
    val charvalue: string -> V (* data invariant: string is of length one *)
    val boolvalue: bool -> V
    val valueint: V -> int
    val valuereal: V -> real
    val valuestring: V -> string
    val valuechar: V -> string
    val valuebool: V -> bool

    val undef: V
    val isundefined: V -> bool

    val empty: V -> bool
    val sametype: V * V -> bool
    val tkString: V -> string
    val Filter: V * (V -> bool) -> V
    val grabElem: V -> V
    val AndReduce: V * (V -> bool) -> V
    val OrReduce: V * (V -> bool) -> V
    val Map: V * (V -> 'a) -> 'a list
    val SetMap: V * (V -> V) -> V
    val rangeset: V * V -> V
end;

structure Value: VALUE =
  struct
  datatype V = undefined
```

```
             | Set of (V list)
             | Sequence of (V list)
             | Tuple of (string * V) list
             | i of int
             | r of real
             | str of string
             | c of string
             | b of bool;
  exception Set_not_homogeneous
  exception Sequence_not_homogeneous
  exception Arg_not_set
  exception Arg_not_sequence
  exception Type_error
  exception Sequence_empty

fun add(i(i1), i(i2)) = i(i1 + i2)
  | add(r(r1), r(r2)) = r(r1 + r2)
  | add(_, _) = raise Type_error;

fun sub(i(i1), i(i2)) = i(i1 - i2)
  | sub(r(r1), r(r2)) = r(r1 - r2)
  | sub(_, _) = raise Type_error;

fun divide(i(i1), i(i2)) = i(i1 div i2)
  | divide(r(r1), r(r2)) = r(r1 / r2)
  | divide(_, _) = raise Type_error;

fun mult(i(i1), i(i2)) = i(i1 * i2)
  | mult(r(r1), r(r2)) = r(r1 * r2)
  | mult(_, _) = raise Type_error;

fun modulo(i(i1), i(i2)) = i(i1 mod i2)
  | modulo(_, _) = raise Type_error;

fun less(i(i1), i(i2)) = b(i1 < i2)
  | less(r(r1), r(r2)) = b(r1 < r2)
  | less(str(s1), str(s2)) = b(s1 < s2)
  | less(c(c1), c(c2)) = b(c1 < c2)
  | less(_, _) = raise Type_error;

fun lesseq(i(i1), i(i2)) = b(i1 <= i2)
  | lesseq(r(r1), r(r2)) = b(r1 <= r2)
  | lesseq(str(s1), str(s2)) = b(s1 <= s2)
  | lesseq(c(c1), c(c2)) = b(c1 <= c2)
  | lesseq(_, _) = raise Type_error;

fun greater(i(i1), i(i2)) = b(i1 > i2)
  | greater(r(r1), r(r2)) = b(r1 > r2)
  | greater(str(s1), str(s2)) = b(s1 > s2)
  | greater(c(c1), c(c2)) = b(c1 > c2)
  | greater(_, _) = raise Type_error;

fun greatereq(i(i1), i(i2)) = b(i1 >= i2)
  | greatereq(r(r1), r(r2)) = b(r1 >= r2)
  | greatereq(str(s1), str(s2)) = b(s1 >= s2)
  | greatereq(c(c1), c(c2)) = b(c1 >= c2)
  | greatereq(_, _) = raise Type_error;

fun andexp(b(b1), b(b2)) = b(b1 andalso b2)
  | andexp(_, _) = raise Type_error;

fun or(b(b1), b(b2)) = b(b1 orelse b2)
  | or(_, _) = raise Type_error;

fun implies(b(b1), b(b2)) = b(not(b1) orelse b2)
  | implies(_, _) = raise Type_error;

fun neg(b(b1)) = b(not b1)
  | neg(_) = raise Type_error;

fun intvalue(in1) = i in1;

fun realvalue(rl1) = r rl1;

fun stringvalue(s1) = str s1;

fun charvalue(c1) =
  if length(explode c1) = 1 then c c1
  else raise Type_error;

fun boolvalue(b1) = b b1;

fun valueint(i i1) = i1
  | valueint(_) = raise Type_error;

fun valuereal(r r1) = r1
  | valuereal(_) = raise Type_error;

fun valuestring(str s1) = s1
  | valuestring(_) = raise Type_error;

fun valuechar(c c1) = c1
  | valuechar(_) = raise Type_error;

fun valuebool(b b1) = b1
  | valuebool(_) = raise Type_error;

val undef = undefined;

fun isundefined(undefined) = true
  | isundefined(_) = false;

fun mkset(x) = Set([x]);

val emptyset = Set([]);

fun subset(Set([]), Set(xs)) = b(true)
  | subset(Set(x::xs), Set(ys)) =
      if valuebool(member(x, Set(ys))) then subset(Set(xs), Set(ys))
      else b(false)
  | subset(_, _) = raise Type_error

and member(undefined, _) = raise Type_error
  | member(_, Set([])) = b(false)
  | member(m, Set(x::xs)) =
      if valuebool(equal(m, x)) then b(true)
      else member(m, Set(xs))
  | member(_, Sequence([])) = b(false)
  | member(m, Sequence(x::xs)) =
      if valuebool(equal(m, x)) then b(true)
      else member(m, Sequence(xs))
  | member(_, _) = raise Type_error

and equal(Sequence([]), Sequence([])) = b(true)
  | equal(Sequence([]), Sequence(_::_)) = b(false)
  | equal(Sequence(_::_), Sequence([])) = b(false)
  | equal(Sequence(x::xs), Sequence(y::ys)) =
      b(valuebool(equal(x,y)) andalso
        valuebool(equal(Sequence(xs), Sequence(ys))))
  | equal(Set(xs), Set(ys)) =
      b(valuebool(subset(Set(xs), Set(ys))) andalso
        valuebool(subset(Set(ys), Set(xs))))
  | equal(Tuple([]), Tuple([])) = b(true)
  | equal(Tuple([]), Tuple(_::_)) = b(false)
  | equal(Tuple(_::_), Tuple([])) = b(false)
  | equal(Tuple((s, t)::ts), Tuple((s1, t1)::ts1)) =
      b(s = s1 andalso valuebool(equal(t, t1)) andalso
        valuebool(equal(Tuple(ts), Tuple(ts1))))
  | equal(r(x), r(y)) = b(x = y)
  | equal(i(x), i(y)) = b(x = y)
  | equal(str(x), str(y)) = b(x = y)
  | equal(c(x), c(y)) = b(x = y)
  | equal((b(x)), (b(y))) = b(x = y)
  | equal(_, _) = raise Type_error;


fun empty(Sequence([])) = true
  | empty(Set([])) = true
  | empty(Sequence(_)) = false
  | empty(Set(_)) = false
  | empty(_) = raise Type_error;

fun sametype((i(x)), (i(y))) = true
  | sametype((r(x)), (r(y))) = true
  | sametype((str(x)), (str(y))) = true
  | sametype(c(_), c(_)) = true
  | sametype((b(x)), (b(y))) = true
  | sametype(Sequence([]), Sequence(_)) = true
  | sametype(Sequence(_), Sequence([])) = true
  | sametype(Sequence(x::xs), Sequence(y::ys)) = sametype(x, y)
  | sametype(Set([]), Set(_)) = true
  | sametype(Set(_), Set([])) = true
  | sametype(Set(x::xs), Set(y::ys)) = sametype(x, y)
  | sametype(Tuple([]), Tuple([])) = true
  | sametype(Tuple([]), Tuple(_::_)) = false
  | sametype(Tuple(_::_), Tuple([])) = false
  | sametype(Tuple((s, t)::ts), Tuple((s1, t1)::ts1)) =
      s = s1 andalso sametype(t, t1) andalso sametype(Tuple(ts), Tuple(ts1))
  | sametype(_, _) = false;
(* note that the condition for sets and sequences is strong enough because
   homogeneity is enforced when they are constructed *)

fun put(x, Set(xs)) = Set(x::xs);

fun unn(Set([]), Set(ys)) = Set(ys)
  | unn(Set(ys), Set([])) = Set(ys)
  | unn(Set(x::xs), Set(ys)) =
      if valuebool(member(x,Set(ys))) then unn(Set(xs), Set(ys))
      else put(x, unn(Set(xs), Set(ys)))
  | unn(_, _) = raise Arg_not_set;

fun union(S1, S2) =
  if sametype(S1, S2) then unn(S1, S2)
  else raise Type_error;

fun intersect(Set([]), Set(ys)) = emptyset
  | intersect(Set(ys), Set([])) = emptyset
  | intersect(Set(x::xs), Set(ys)) =
      if valuebool(member(x,Set(ys))) then put(x, intersect(Set(xs), Set(ys)))
      else intersect(Set(xs), Set(ys))
  | intersect(_, _) = raise Arg_not_set;

fun intersection(S1, S2) =
  if sametype(S1, S2) then intersect(S1, S2)
  else raise Type_error;

fun diff(Set([]), Set(ys)) = emptyset
  | diff(Set(ys), Set([])) = Set(ys)
  | diff(Set(x::xs), Set(ys)) =
      if valuebool(member(x,Set(ys))) then diff(Set(xs), Set(ys))
      else put(x, diff(Set(xs), Set(ys)))
  | diff(_, _) = raise Arg_not_set;

fun difference(S1, S2) =
  if sametype(S1, S2) then diff(S1, S2)
  else raise Type_error;

fun size(Set([])) = i 0
  | size(Set(_::xs)) = add(i(1), size(Set (xs)))
  | size(_) = raise Arg_not_set;

fun mkseq x = Sequence([x]);

val emptyseq = Sequence([]);

fun app(Sequence(x), Sequence(y)) = Sequence(x @ y)
```

```
  | app(str(x), str(y)) = str(x^y)
  | app(_, _) = raise Arg_not_sequence;

fun append(S1, S2) =
  if sametype(S1, S2) then app(S1, S2)
  else raise Type_error;

(* the rest of this sequence operators need to be extended to strings
   sometime *)

fun first(Sequence(x::_)) = x
  | first(Sequence([])) = raise Sequence_empty
  | first(_) = raise Arg_not_sequence;

fun header(Sequence(_::[])) = emptyseq
  | header(Sequence(x::xs)) = append(Sequence([x]), header(Sequence (xs)))
  | header(Sequence([])) = raise Sequence_empty
  | header(_) = raise Arg_not_sequence;

fun last(Sequence(x::[])) = x
  | last(Sequence(_::xs)) = last(Sequence (xs))
  | last(Sequence([])) = raise Sequence_empty
  | last(_) = raise Arg_not_sequence;

fun trailer(Sequence (_::xs)) = Sequence (xs)
  | trailer(Sequence([])) = raise Sequence_empty
  | trailer(_) = raise Arg_not_sequence;

fun length(Sequence([])) = i(0)
  | length(Sequence(_::xs)) = add(i(1), length(Sequence(xs)))
  | length(str(s)) = i(String.size s)
  | length(_) = raise Arg_not_sequence;

fun index(Sequence(x::xs), ind) =
      if valueint(ind) = 1 then x
      else index(Sequence (xs), sub(ind, intvalue 1))
  | index(Sequence([]), _) = raise Sequence_empty
  | index(_, _) = raise Arg_not_sequence;

fun mktuple(S) = Tuple(S);

fun field(s, Tuple((s1, v)::svl)) =
      if s = s1 then v
      else field(s, Tuple(svl))
  | field(_, _) = raise Type_error;

fun Filter(Set(elems), pred) = Set(filter(elems, pred))
  | Filter(Sequence(elems), pred) = Set(filter(elems, pred))
  | Filter(_, _) = raise Type_error;

fun grabElem(Set(e::el)) = e
  | grabElem(_) = raise Type_error;

fun tkString(t) =
  let fun sString([]) = ""
        | sString([e]) = tkString(e)
        | sString(e::es) = tkString(e)^", "^sString(es)

      and tString([]) = ""
        | tString([(_, t)]) = tkString(t)
        | tString((_, t)::ts) = tkString(t)^", "^tString(ts)
  in
  case t of
     i(i1) => makestring i1
   | r(r1) => makestring r1
   | str(s1) => s1
   | c(c1) => c1
   | b(b1) => makestring b1
   | Set(s) => "{"^sString(s)^"}"
   | Sequence(s) => "<"^sString(s)^">"
   | Tuple(ts) => "("^tString(ts)^")"
   | undefined => "undefined"
end;

 fun AndReduce(Set(es), pred) = b(andred(es, pred))
   | AndReduce(Sequence(es), pred) = b(andred(es, pred));

 fun OrReduce(Set(es), pred) = b(orred(es, pred))
   | OrReduce(Sequence(es), pred) = b(orred(es, pred));

 fun Map(Set(es), pred) = map pred es
   | Map(Sequence(es), pred) = map pred es;

 fun SetMap(Set(es), pred) = Set(Map(Set(es), pred));

 fun rangeset(low, high) =
   if valuebool(less(high, low)) then emptyset
   else union(mkset(low), rangeset(add(low, intvalue 1), high));

end;

datatype spectype = inttype
                  | realtype
                  | chartype
                  | stringtype
                  | booltype
                  | voidtype
                  | settype of spectype
                  | seqtype of spectype
                  | tupletype of (string * spectype) list
                  | alttype of spectype list
                  | typename of string
                  | reftype of spectype;

datatype expr =
                 primi of int
               | primr of real
               | prims of string
               | primb of bool
               | primc of string
               | add of (expr * expr)
               | sub of (expr * expr)
               | divide of (expr * expr)
               | mult of (expr * expr)
               | modulo of (expr * expr)
               | buildset of (expr list)
               | union of (expr * expr)
               | intersection of (expr * expr)
               | difference of (expr * expr)
               | member of (expr * expr)
               | subset of (expr * expr)
               | size of expr
               | buildseq of (expr list)
               | append of (expr * expr)
               | first of expr
               | header of expr
               | last of expr
               | trailer of expr
               | length of expr
               | index of (expr * expr)
               | buildtuple of ((string * expr) list)
               | field of (string * expr)
               | less of (expr * expr)
               | lesseq of (expr * expr)
               | greater of (expr * expr)
               | greatereq of (expr * expr)
               | eq of (expr * expr)
               | andexp of (expr * expr)
               | or of (expr * expr)
               | implies of (expr * expr)
               | neg of expr
               | ident of string
               | primed of string
               | result
               | forall of (string * expr * expr * expr)
               | exists of (string * expr * expr)
               | setcomp of (string * expr * expr * expr)
               | subrange of (expr * expr)
               | isoftype of (expr * spectype)
               | call of (string * expr list)
               | mcall of (expr * string * expr list)
;

fun noprimed(add(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(sub(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(divide(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(mult(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(modulo(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(buildset(el)) = andred(el, noprimed)
  | noprimed(setcomp(_, dom, e, pred)) = andred([dom, e, pred], noprimed)
  | noprimed(union(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(intersection(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(difference(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(size(e)) = noprimed(e)
  | noprimed(buildseq(el)) = andred(el, noprimed)
  | noprimed(append(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(first(e)) = noprimed(e)
  | noprimed(header(e)) = noprimed(e)
  | noprimed(last(e)) = noprimed(e)
  | noprimed(trailer(e)) = noprimed(e)
  | noprimed(length(e)) = noprimed(e)
  | noprimed(index(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(buildtuple(el)) = andred(el, fn(s, e) => noprimed(e))
  | noprimed(field(_, e)) = noprimed(e)
  | noprimed(primed(_)) = false
  | noprimed(subrange(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(less(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(lesseq(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(greater(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(greatereq(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(eq(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(member(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(subset(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(andexp(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(or(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(implies(e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(neg(e)) = noprimed(e)
  | noprimed(forall(_, e1, e2, e3)) = andred([e1, e2, e3], noprimed)
  | noprimed(exists(_, e1, e2)) = noprimed(e1) andalso noprimed(e2)
  | noprimed(isoftype(e, _)) = noprimed(e)
  | noprimed(call(_, el)) = andred(el, noprimed)
  | noprimed(mcall(instance, _, args)) = andred(instance::args, noprimed)
  | noprimed(result) = false
  | noprimed(_) = true;

datatype id = bound of string | unbound of string;

signature ENV =
  sig
  type E
  exception Not_found of id
  val empty: E
  val extend: E * id * Value.V * spectype -> E
  val replace: E * id * Value.V * spectype -> E
  val lookup: E * id -> Value.V
  val getType: E * id -> spectype
  val isbound: E * id -> bool
  val append: E * E -> E
end;

structure Env: ENV =
  struct
  type E = (id * Value.V * spectype) list;
  exception Not_found of id;

  val empty = [];
  fun extend(env, name, v, t) = (name, v, t)::env;

  fun replace([], name, v, t) = raise Not_found name
    | replace((n, v, t)::env, name, va, ty) =
        if n = name then (name, va, ty)::env
        else (n, v, t)::replace(env, name, va, ty);

  fun lookup([], n1) = raise Not_found n1
    | lookup((n, v, _)::env, n1) =
        if n = n1 then v
        else lookup(env, n1);

  fun getType([], n1) = raise Not_found n1
    | getType((n, _, t)::env, n1) =
        if n = n1 then t
        else getType(env, n1);

  fun isbound([], _) = false
    | isbound((n, _, _)::env, n1) = if n = n1 then true
                                    else isbound(env, n1);

  fun append(e1, e2) = e1 @ e2;

end;

signature TENV =
  sig
  type E
  exception Not_found of string
  val empty: E
  val extend: E * string * spectype -> E
  val lookup: E * string -> spectype
  val isbound: E * string -> bool
  val getpos: E * string -> int
end;

structure TEnv: TENV =
  struct
  type E = (string * spectype) list;
  exception Not_found of string;

  val empty = [];

  fun extend(env, name, t) = (name, t)::env;

  fun lookup([], name) = raise Not_found(name)
    | lookup((n, t)::env, n1) =
        if n = n1 then t
        else lookup(env, n1);

  fun isbound([], name) = false
    | isbound((n, t)::env, name) = (n = name) orelse isbound(env, name);
  fun getpos([], name) = raise Not_found(name)
    | getpos((n, t)::env, name) =
        if n = name then 1
        else 1 + getpos(env, name);
end;

type absfunctiontype = {name: string,
                        args: TEnv.E,
                        returntype: spectype,
                        body: expr};

datatype operationtype = operation of {name: string,
                                       args: TEnv.E,
                                       returntype: spectype,
                                       modifies: string list,
                                       pre: expr,
                                       post: expr}
                       | friend of {name: string,
                                    friendclass: string,
                                    args: TEnv.E,
                                    returntype: spectype,
                                    modifies: string list,
                                    pre: expr,
                                    post: expr};


datatype parentclasstype = none | parentclassname of string;

type classspec = {name: string,
                  parentclass: parentclasstype,
                  invariant: expr,
                  absfunctions: absfunctiontype list,
                  publicops: operationtype list,
                  protectedops: operationtype list};

type spec = {classes: classspec list,
             typeenv: TEnv.E};

(* ADT type *)

 fun chasetype(typename(s), tenv) =
      (case TEnv.lookup(tenv, s) of
         typename(s1) => chasetype(typename(s1), tenv)
       | t => t)
   | chasetype(reftype(t), tenv) = chasetype(t, tenv)
   | chasetype(t, tenv) = t;

fun gettuptype(s, tupletype((s1, t)::tl)) =
      if s = s1 then t else gettuptype(s, tupletype(tl))
  | gettuptype(s, _) = raise Value.Type_error;

fun getseqtype(seqtype(it)) = it
  | getseqtype(_) = raise Value.Type_error;
```

```
fun findtype(_, alttype([]), _) = raise Value.Type_error
  | findtype(lv, alttype(t::tlist), tenv) =
      (findtype(lv, t, tenv)
       handle _ => findtype(lv, alttype(tlist), tenv))
  | findtype(primed(_), t, tenv) = t
  | findtype(field(s, lv), t, tenv) = chasetype(gettuptype(s, findtype(lv, t,
                                                                      tenv)),
                                                tenv)
  | findtype(index(lv, _), t, tenv) = chasetype(getseqtype(findtype(lv, t,
                                                                    tenv)),
                                                tenv)
  | findtype(first(lv), t, tenv) = chasetype(getseqtype(findtype(lv, t, tenv)),
                                              tenv)
  | findtype(last(lv), t, tenv) = chasetype(getseqtype(findtype(lv, t, tenv)),
                                             tenv)
  | findtype(header(lv), t, tenv) = findtype(lv, t, tenv)
  | findtype(trailer(lv), t, tenv) = findtype(lv, t, tenv);

fun sametype(settype(pt), settype(rt), tenv) =
      sametype(chasetype(pt, tenv), chasetype(rt, tenv), tenv)
  | sametype(seqtype(pt), seqtype(rt), tenv) =
      sametype(chasetype(pt, tenv), chasetype(rt, tenv), tenv)
  | sametype(tupletype([]), tupletype([]), _) = true
  | sametype(tupletype((s, pt)::ptl), tupletype((s1, rt)::rtl), tenv) =
      s = s1 andalso sametype(chasetype(pt, tenv), chasetype(rt, tenv), tenv)
      andalso sametype(tupletype(ptl), tupletype(rtl), tenv)
  | sametype(alttype(tl), rt, tenv) =
      orred(tl, fn(t) =>
                 sametype(chasetype(t, tenv), chasetype(rt, tenv), tenv))
  | sametype(pt, alttype(tl), tenv) =
      orred(tl, fn(t) =>
                 sametype(chasetype(t, tenv), chasetype(pt, tenv), tenv))
  | sametype(pt, rt, _) = pt = rt;

fun typeok(pt, lv, rt, tenv) = sametype(pt, findtype(lv, rt, tenv), tenv);

fun typematch(formals, actuals, tenv) =
 let val ftypes = map (fn(s, t) => t) formals in
  len ftypes = len actuals
  (* andalso orred(map *)
 end;

(* end ADT type *)

(* ADT AbsFunction *)

fun getAName({name = n, ...}:absfunctiontype) = n;

fun getAArgs({args = a, ...}:absfunctiontype) = a;

fun getARetType({returntype = r, ...}:absfunctiontype) = r;

fun getABody({body = b, ...}:absfunctiontype) = b;
```

```
(* end ADT AbsFunction *)

(* ADT Spec *)

fun getTEnv({typeenv = t, ...}:spec) = t;

fun getClass(typename(tname), {classes = c, ...}:spec) =
      find(c, fn({name = n, ...}) => n = tname);

fun isClass(typename(tname), {classes = c, ...}:spec) =
      orred(c, fn({name = n, ...}) => n = tname)
  | isClass(_, _) = false;

fun getFriend(opname, args, {classes = c, ...}:spec) =
     find(reduce((map (fn({publicops = p, ...}) => p) c), op @, []),
          (fn(opn) => case opn of
                       friend({name = n, ...}) => n = opname
                     | operation(_) => false));

fun isAbsFunction(name, numargs, {classes = c, ...}:spec) =
     orred(reduce((map (fn({absfunctions = f, ...}) => f) c), op @, []),
           (fn(ab) => name = getAName(ab) andalso numargs = len(getAArgs(ab))));

fun getAbsFunction(name, {classes = c, ...}:spec) =
     find(reduce((map (fn({absfunctions = f, ...}) => f) c), op @, []),
          (fn(ab) => name = getAName(ab)));

fun isfriend(oper) = oper = "";

(* end ADT Spec *)

(* ADT Operation *)

fun getName(operation({name = n, ...})) = n
  | getName(friend({name = n, ...})) = n;

fun getPre(operation({pre = p, ...})) = p
  | getPre(friend({pre = p, ...})) = p;

fun getPost(operation({post = p, ...})) = p
  | getPost(friend({post = p, ...})) = p;

fun getModifies(operation({modifies = m, ...})) = m
  | getModifies(friend({modifies = m, ...})) = m;

fun getArgs(operation({args = a, ...})) = a
  | getArgs(friend({args = a, ...})) = a;

fun fixForms(forms, object, typ, rettype) =
    let val tenv = if isfriend(object) then forms
                   else TEnv.extend(forms, "self", typ) in
      if rettype = voidtype then tenv
      else TEnv.extend(tenv, "result", rettype) end;
```

```
fun fixArgs(acts, object, typ, rettype) =
    let val ac = if isfriend(object) then acts
                 else ident(object)::acts in
      if rettype = voidtype then ac
      else (primi 0)::ac end;

fun getRetType(operation({returntype = rt, ...})) = rt
  | getRetType(friend({returntype = rt, ...})) =  rt;

(* end of ADT Operation *)

(* ADT Class *)

fun getOp(opname, {publicops = ol, ...}:classspec) =
(* this must change when inheritance is added *)
      find(ol, fn(opn) => getName(opn) = opname);

fun getInv({invariant = i, ...}:classspec) = i;

(* end of ADT Class *)

fun lhs(primed(_)) = true
  | lhs(result) = true
  | lhs(field(_, e)) = lhs(e)
  | lhs(index(e1, e2)) = lhs(e1) andalso noprimed(e2)
  | lhs(first(e)) = lhs(e)
  | lhs(last(e)) = lhs(e)
  | lhs(header(e)) = lhs(e)
  | lhs(trailer(e)) = lhs(e)
  | lhs(e) = noprimed(e);

fun constructive(eq(e1, e2)) = (lhs(e1) andalso noprimed(e2)) orelse
                               (lhs(e2) andalso noprimed(e1))
  | constructive(member(e1, e2)) = noprimed(e1) andalso lhs(e2)
  | constructive(subset(e1, e2)) = noprimed(e1) andalso lhs(e2)
  | constructive(neg(_)) = false
  | constructive(forall(_, dom, antecedent, consequent)) =
      noprimed(dom) andalso noprimed(antecedent)
      andalso constructive(consequent)
  | constructive(exists(_, dom, conjunct)) =
      noprimed(dom) andalso constructive(conjunct)
  | constructive(primed(_)) = false
  | constructive(result) = false
  | constructive(andexp(e1, e2)) = constructive(e1) andalso constructive(e2)
  | constructive(or(_, _)) = false
  | constructive(implies(antecedent, consequent)) =
      noprimed(antecedent) andalso constructive(consequent)
  | constructive(e) = noprimed(e);

(* given an lvalue and a Value.V, return the value that would be given by
   replacing the primed(_) expr in the lvalue with the Value.V.
   Not well tested yet. *)

fun construct(primed(_), v) = v
```

```
  | construct(field(s, e), v) = Value.field(s, construct(e, v))
  | construct(index(e, primi x), v) = Value.index(construct(e, v),
                                                  Value.intvalue x)
  | construct(first(e), v) = Value.first(construct(e, v))
  | construct(last(e), v) = Value.last(construct(e, v))
  | construct(header(e), v) = Value.header(construct(e, v))
  | construct(trailer(e), v) = Value.trailer(construct(e, v))
;

datatype constraint = ceq of expr * Value.V
                    | cmember of Value.V * expr
                    | csubset of Value.V * expr;

fun extractlhs(ceq(e, _)) = e
  | extractlhs(cmember(_, e)) = e
  | extractlhs(csubset(_, e)) = e;

fun match(c, lvalue) = lvalue = extractlhs(c);

fun partialmatch(c, lvalue) =
  let fun helpmatch(l) =
    if l = lvalue then true
    else
      case l of
        field(_, l1) => helpmatch(l1)
      | index(l1, _) => helpmatch(l1)
      | first(l1) => helpmatch(l1)
      | last(l1) => helpmatch(l1)
      | header(l1) => helpmatch(l1)
      | trailer(l1) => helpmatch(l1)
      | _ => false
  in helpmatch(extractlhs(c)) end;

fun isconsbound(clist, lvalue, pred) =
      orred(clist, fn(elem) => pred(elem, lvalue));

fun replace(_, _, []) = []
  | replace(pat, new, c::clist) =
     if partialmatch(c, pat) then
       let fun replaceaux(pat, new, l) =
         if pat = l then new
         else case l of
                field(s, l1) => field(s, replaceaux(pat, new, l1))
              | index(l1, ind) => index(replaceaux(pat, new, l1), ind)
              | first(l1) => first(replaceaux(pat, new, l1))
              | last(l1) => last(replaceaux(pat, new, l1))
              | header(l1) => header(replaceaux(pat, new, l1))
              | trailer(l1) => trailer(replaceaux(pat, new, l1))
       in case c of
            ceq(l, v) => ceq(replaceaux(pat, new, l), v)
                         ::replace(pat, new, clist)
          | cmember(v, l) => cmember(v, replaceaux(pat, new, l))
                             ::replace(pat, new, clist)
          | csubset(v, l) => csubset(v, replaceaux(pat, new, l))
```

```
                              ::replace(pat, new, clist)
       end
     else c::replace(pat, new, clist);

fun isid(ident(_)) = true
  | isid(primed(_)) = true
  | isid(_) = false;

fun getident(ident(s)) = s
  | getident(primed(s)) = s;

 fun makeNewEnv(env, object, newenv, formals, actuals, modslist) =
  let fun mne(modslist, env) =
   case (modslist) of
     [] =>
       let val d = if Env.isbound(newenv, unbound("result")) then
                   output(std_out,
                          Value.tkString(Env.lookup(newenv,
                                                    unbound("result")))^"\n")
                   else output(std_out, "\n")
       in env end
    | m::mlist =>
       if m = "result" then mne(mlist, env)
       else if m = "self" then
         mne(mlist,
             Env.replace(env, bound(object),
                         Env.lookup(newenv, unbound("self")),
                         Env.getType(newenv, unbound("self"))))
       else if not(TEnv.isbound(formals, m)) then
         mne(mlist,
             Env.replace(env, bound(m),
                         Env.lookup(newenv, unbound(m)),
                         Env.getType(newenv, unbound(m))))
       else let val actual = nth(actuals, TEnv.getpos(formals, m) - 1) in
         if isid(actual)
           then mne(mlist,
                    Env.replace(env, bound(getident(actual)),
                                Env.lookup(newenv, unbound(m)),
                                Env.getType(newenv, unbound(m))))
           else mne(mlist, env) end
   in mne(modslist, env) end;

fun composeset([]) = Value.emptyset
  | composeset(cmember(v, _)::clist) = Value.union(Value.mkset(v),
                                                    composeset(clist))
  | composeset(csubset(vs, _)::clist) = Value.union(vs, composeset(clist));

exception Not_implemented;
exception Insufficiently_constructive of string;

fun getconsvalue(name, t, constraints, mutating, env, tenv) =
  let fun buildval(t, lvalue, constraints) =
    if isconsbound(constraints, lvalue, match) then
      case filter(constraints, fn(elem) => match(elem, lvalue)) of
```

```
        [ceq(_, v)] => v
      | clist => composeset(clist)
      else
        case t of
             tupletype(fieldlist) =>
               Value.mktuple(map (fn(s, ty) => (s, buildval(chasetype(ty, tenv),
                                                             field(s, lvalue),
                                                             constraints)))
              (* could filter the constraints here to improve efficiency *)
                             fieldlist)
             | seqtype(itemtype) =>
                 if isconsbound(constraints, header(lvalue), partialmatch)
                   then let val newheader = buildval(t, header(lvalue), constraints)
                            val len = 1 + Value.valueint(Value.length(newheader)) in
                       Value.append(newheader,
                         Value.mkseq(buildval(chasetype(itemtype, tenv),
                                              index(lvalue, primi len),
                                              replace(last(lvalue),
                                                      index(lvalue, primi len),
                                                      constraints)))) end
                 else if isconsbound(constraints, trailer(lvalue), partialmatch) then
                   Value.append(Value.mkseq(buildval(chasetype(itemtype, tenv),
                                                     index(lvalue, primi 1),
                                                     constraints)),
                                buildval(t, trailer(lvalue), constraints))
                 else composeseq(chasetype(itemtype, tenv), lvalue, constraints, 1)
             | alttype([]) => raise Insufficiently_constructive(name)
             | alttype(ty::tlist) =>
                 (buildval(chasetype(ty, tenv), lvalue, constraints)
                  handle _ => buildval(alttype(tlist), lvalue, constraints))
        | _ => if mutating andalso typeok(chasetype(t, tenv), lvalue,
                                          chasetype(Env.getType(env,
                                                                bound(name)),
                                                    tenv), tenv)
                 then construct(lvalue, Env.lookup(env, bound(name)))
                 else raise Insufficiently_constructive(name)
    and  composeseq(t, lvalue, constraints, ind) =
             if isconsbound(constraints, index(lvalue, primi ind), partialmatch) then
               Value.append(Value.mkseq(buildval(t, index(lvalue, primi ind),
                                                 constraints)),
                            composeseq(t, lvalue, constraints, ind + 1))
               else if mutating andalso
                       Value.valueint(
                        Value.length(
                         construct(lvalue, Env.lookup(env, bound(name))))) >= ind then
                 Value.append(Value.mkseq(construct(index(lvalue, primi ind),
                                                    Env.lookup(env, bound(name)))),
                              composeseq(t, lvalue, constraints, ind + 1))
               else Value.emptyseq
  in buildval(t, primed(name), constraints) end;

fun constraintstoenv(constraints, mods, params, env, tenv) =
  case mods of
    [] => Env.empty
  | m::modslist =>
      let val typ = if TEnv.isbound(params, m)
                    then TEnv.lookup(params, m)
                    else Env.getType(env, bound(m)) in
      Env.extend(constraintstoenv(constraints, modslist, params, env, tenv),
                 unbound(m),
                 getconsvalue(m, chasetype(typ, tenv), constraints,
                              not(Value.isundefined(Env.lookup(env, bound(m)))),
                              env, tenv),
                 typ) end;

 exception Precondition_not_satisfied;
 exception Postcondition_not_satisfied;
 exception Wrong_number_of_args;

fun plhs(e) = lhs(e) andalso not(noprimed(e));

fun bind(env, spec, [], []) = Env.empty
  | bind(env, spec, (name, typ)::formals, a::actuals) =
    let val v = npeval(a, env, spec)
    in
      Env.extend(bind(env, spec, formals, actuals), bound(name), v, typ) end
  | bind(_, _, _, _) = raise Wrong_number_of_args
and
    npeval(e, env, s) =
 let fun eval(e, env) =
   case e of
     primi i => Value.intvalue i
   | primr r => Value.realvalue r
   | prims s => Value.stringvalue s
   | primc c => Value.charvalue c
   | primb b => Value.boolvalue b
   | add(e1, e2) => Value.add(eval(e1, env), eval(e2, env))
   | sub(e1, e2) => Value.sub(eval(e1, env), eval(e2, env))
   | divide(e1, e2) => Value.divide(eval(e1, env), eval(e2, env))
   | mult(e1, e2) => Value.mult(eval(e1, env), eval(e2, env))
   | modulo(e1, e2) => Value.modulo(eval(e1, env), eval(e2, env))
   | buildset([]) => Value.emptyset
   | buildset(e::el) => Value.union(Value.mkset(eval(e, env)),
                                    eval(buildset(el), env))
   | union(e1, e2) => Value.union(eval(e1, env), eval(e2, env))
   | intersection(e1, e2) => Value.intersection(eval(e1, env), eval(e2, env))
   | difference(e1, e2) => Value.difference(eval(e1, env), eval(e2, env))
   | size(e) => Value.size(eval(e, env))
   | buildseq([]) => Value.emptyseq
   | buildseq(e::el) => Value.append(Value.mkseq(eval(e, env)),
                                     eval(buildseq(el), env))
   | append(e1, e2) => Value.append(eval(e1, env), eval(e2, env))
   | first(e) => Value.first(eval(e, env))
   | header(e) => Value.header(eval(e, env))
   | last(e) => Value.last(eval(e, env))
   | trailer(e) => Value.trailer(eval(e, env))
   | length(e) => Value.length(eval(e, env))
   | index(e1, e2) => Value.index(eval(e1, env), eval(e2, env))
   | buildtuple(el) => Value.mktuple(map (fn(s, e) => (s, eval(e, env))) el)
   | field(s, e) => Value.field(s, eval(e, env))
   | ident(s) => Env.lookup(env, bound(s))
   | subrange(e1, e2) => Value.rangeset(eval(e1, env), eval(e2, env))
   | less(e1, e2) => if not(noprimed(e1) andalso noprimed(e2))
                     then Value.boolvalue(true)
                     else Value.less(eval(e1, env), eval(e2, env))
   | lesseq(e1, e2) => if not(noprimed(e1) andalso noprimed(e2))
                     then Value.boolvalue(true)
                     else Value.lesseq(eval(e1, env), eval(e2, env))
   | greater(e1, e2) => if not(noprimed(e1) andalso noprimed(e2))
                     then Value.boolvalue(true)
                     else Value.greater(eval(e1, env), eval(e2, env))
   | greatereq(e1, e2) => if not(noprimed(e1) andalso noprimed(e2))
                     then Value.boolvalue(true)
                     else Value.greatereq(eval(e1, env), eval(e2, env))
   | eq(e1, e2) => if not(noprimed(e1) andalso noprimed(e2))
                     then Value.boolvalue(true)
                     else Value.equal(eval(e1, env), eval(e2, env))
   | member(e1, e2) => if not(noprimed(e1) andalso noprimed(e2))
                     then Value.boolvalue(true)
                     else Value.member(eval(e1, env), eval(e2, env))
   | subset(e1, e2) => if not(noprimed(e1) andalso noprimed(e2))
                     then Value.boolvalue(true)
                     else Value.subset(eval(e1, env), eval(e2, env))
   | andexp(e1, e2) => Value.andexp(eval(e1, env), eval(e2, env))
   | or(e1, e2) => Value.or(eval(e1, env), eval(e2, env))
   | implies(e1, e2) => if not(noprimed(e1) andalso noprimed(e2))
                     then Value.boolvalue(true)
                     else Value.implies(eval(e1, env), eval(e2, env))
   | neg(e) => if not(noprimed(e)) then Value.boolvalue(true)
               else Value.neg(eval(e, env))
   | forall(x, dom, antecedent, consequent) =>
       if not(noprimed(dom))
         then Value.boolvalue(true)
         else Value.AndReduce(eval(dom, env),
             fn(elem) => Value.valuebool(eval(implies(antecedent, consequent),
                                     Env.extend(env, bound(x),
                                                elem, tupletype([])))))
(* note that the type needs to be fixed for type inferencing *)
   | exists(x, dom, conjunct) =>
       if not(noprimed(dom))
         then Value.boolvalue(true)
         else Value.OrReduce(eval(dom, env),
             fn(elem) => Value.valuebool(eval(conjunct,
                                     Env.extend(env, bound(x),
                                                elem, tupletype([])))))
   | setcomp(x, dom, e, pred) =>
       Value.SetMap(
         Value.Filter(eval(dom, env),
           fn(elem) => Value.valuebool(eval(pred,
                                   Env.extend(env, bound(x),
                                              elem, tupletype([]))))),
         fn(elem) => eval(e, Env.extend(env, bound(x), elem, tupletype([]))))
```

```
    | isoftype(e, t) => raise Not_implemented
    | call(name, args) => funcall(name, args, env, s)
    | mcall(obj, name, args) => methcall(obj, name, args, env, s)
  in eval(e, env) end
and
    getCons(andexp(e1, e2), env, s) =
  let val r1 = getCons(e1, env, s)
      val r2 = getCons(e2, env, s) in
    (fst(r1) @ fst(r2), snd(r1) @ snd(r2))
  end
  | getCons(or(e1, e2), env, s) =
    if not(Value.valuebool(npeval(e1, env, s))) then getCons(e2, env, s)
    else if not(Value.valuebool(npeval(e2, env, s))) then getCons(e1, env, s)
    else ([], [or(e1, e2)])
  | getCons(implies(e1, e2), env, s) =
    if noprimed(e1) then
      if Value.valuebool(npeval(e1, env, s)) then
        getCons(e2, env, s)
        else ([], [])
    else ([], [implies(e1, e2)])
  | getCons(e, _, _) = if constructive(e) then ([e], [])
                                       else ([], [e])
and
    beval(e, env, s) =
 let fun eval(e, env) =
   case e of
     primi i => Value.intvalue i
   | primr r => Value.realvalue r
   | prims s => Value.stringvalue s
   | primc c => Value.charvalue c
   | primb b => Value.boolvalue b
   | add(e1, e2) => Value.add(eval(e1, env), eval(e2, env))
   | sub(e1, e2) => Value.sub(eval(e1, env), eval(e2, env))
   | divide(e1, e2) => Value.divide(eval(e1, env), eval(e2, env))
   | mult(e1, e2) => Value.mult(eval(e1, env), eval(e2, env))
   | modulo(e1, e2) => Value.modulo(eval(e1, env), eval(e2, env))
   | buildset([]) => Value.emptyset
   | buildset(e::el) => Value.union(Value.mkset(eval(e, env)),
                                    eval(buildset(el), env))
   | union(e1, e2) => Value.union(eval(e1, env), eval(e2, env))
   | intersection(e1, e2) => Value.intersection(eval(e1, env), eval(e2, env))
   | difference(e1, e2) => Value.difference(eval(e1, env), eval(e2, env))
   | size(e) => Value.size(eval(e, env))
   | buildseq([]) => Value.emptyseq
   | buildseq(e::el) => Value.append(Value.mkseq(eval(e, env)),
                                     eval(buildseq(el), env))
   | append(e1, e2) => Value.append(eval(e1, env), eval(e2, env))
   | first(e) => Value.first(eval(e, env))
   | header(e) => Value.header(eval(e, env))
   | last(e) => Value.last(eval(e, env))
   | trailer(e) => Value.trailer(eval(e, env))
   | length(e) => Value.length(eval(e, env))
   | index(e1, e2) => Value.index(eval(e1, env), eval(e2, env))
   | buildtuple(el) => Value.mktuple(map (fn(s, e) => (s, eval(e, env))) el)
```

```
    | field(s, e) => Value.field(s, eval(e, env))
    | ident(s) => Env.lookup(env, bound(s))
    | subrange(e1, e2) => Value.rangeset(eval(e1, env), eval(e2, env))
    | less(e1, e2) => Value.less(eval(e1, env), eval(e2, env))
    | lesseq(e1, e2) => Value.lesseq(eval(e1, env), eval(e2, env))
    | greater(e1, e2) => Value.greater(eval(e1, env), eval(e2, env))
    | greatereq(e1, e2) => Value.greatereq(eval(e1, env), eval(e2, env))
    | eq(e1, e2) => Value.equal(eval(e1, env), eval(e2, env))
    | member(e1, e2) => Value.member(eval(e1, env), eval(e2, env))
    | subset(e1, e2) => Value.subset(eval(e1, env), eval(e2, env))
    | andexp(e1, e2) => Value.andexp(eval(e1, env), eval(e2, env))
    | or(e1, e2) => Value.or(eval(e1, env), eval(e2, env))
    | implies(e1, e2) => Value.implies(eval(e1, env), eval(e2, env))
    | neg(e) => Value.neg(eval(e, env))
    | forall(x, dom, antecedent, consequent) =>
        Value.AndReduce(eval(dom, env),
              fn(elem) => Value.valuebool(eval(implies(antecedent, consequent),
                                                Env.extend(env, bound(x), elem,
                                                           tupletype([])))))
    | exists(x, dom, e) =>
        Value.OrReduce(eval(dom, env),
              fn(elem) => Value.valuebool(eval(e,
                                                Env.extend(env, bound(x), elem,
                                                           tupletype([])))))
    | setcomp(x, dom, e, pred) =>
        Value.SetMap(
          Value.Filter(eval(dom, env),
            fn(elem) => Value.valuebool(eval(pred,
                                              Env.extend(env, bound(x), elem,
                                                         tupletype([]))))),
          fn(elem) => eval(e, Env.extend(env, bound(x), elem, tupletype([]))))
    | isoftype(e, t) => raise Not_implemented
    | call(n, el) => funcall(n, el, env, s)
    | mcall(obj, n, args) => methcall(obj, n, args, env, s)
    | primed(s) => Env.lookup(env, unbound(s))
    | result => Env.lookup(env, unbound("result"))
   in eval(e, env) end
and

(* evaluate all the indices in index exprs in lvalues, convert the result
   constructor for exprs to primed("result") so that it can be stored in the
   environment, and replace instances of first with index(_, 1) *)

    evalindex(index(e1, e2), env, s) =
      index(evalindex(e1, env, s), primi(Value.valueint(npeval(e2, env, s))))
  | evalindex(field(n, e), env, s) = field(n, evalindex(e, env, s))
  | evalindex(first(e), env, s) = index(evalindex(e, env, s), primi 1)
  | evalindex(last(e), env, s) = last(evalindex(e, env, s))
  | evalindex(header(e), env, s) = header(evalindex(e, env, s))
  | evalindex(trailer(e), env, s) = trailer(evalindex(e, env, s))
  | evalindex(result, _, _) = primed("result")
  | evalindex(e, _, _) = e
and
    feval(el, env, s) =
 let fun eval(e, env) =
   case e of
     eq(e1, e2) => if plhs(e1) then [ceq(evalindex(e1, env, s),
                                         npeval(e2, env, s))]
                   else if plhs(e2) then [ceq(evalindex(e2, env, s),
                                              npeval(e1, env, s))]
                   else []
   | member(e1, e2) => if plhs(e2) then [cmember(npeval(e1, env, s),
                                                 evalindex(e2, env, s))]
                       else []
   | subset(e1, e2) => if plhs(e2) then [csubset(npeval(e1, env, s),
                                                 evalindex(e2, env, s))]
                       else []
   | andexp(e1, e2) => eval(e1, env) @ eval(e2, env)
   | implies(e1, e2) => if Value.valuebool(npeval(e1, env, s))
                          then eval(e2, env)
                          else []
   | forall(x, dom, antecedent, consequent) =>
       reduce(Value.Map(npeval(dom, env, s),
                        fn(elem) => eval(implies(antecedent, consequent),
                                         Env.extend(env, bound(x), elem,
                                                    tupletype([])))),
              op @, [])
   | exists(x, dom, conjunct) =>
       eval(conjunct,
            Env.extend(env, bound(x),
                       Value.grabElem(
                         Value.Filter(npeval(dom, env, s),
                           fn(elem) =>
                             Value.valuebool(
                               npeval(conjunct,
                                      Env.extend(env, bound(x), elem,
                                                 tupletype([])),
                                      s)))),
                       tupletype([])))
   | _ => []
  in reduce((map (fn(e) => eval(e, env)) el), op @, []) end
and
    geneval(pre, post, formals, actuals, modslist, env, spec) =
  let val benv = Env.append(bind(env, spec, formals, actuals), env) in
    if not(Value.valuebool(npeval(pre, benv, spec)))
      then raise Precondition_not_satisfied
    else if not(Value.valuebool(npeval(post, benv, spec)))
      then raise Postcondition_not_satisfied
    else let val newenv =
     constraintstoenv(feval(fst(getCons(post, benv, spec)), benv, spec),
                      modslist, formals, benv, getTEnv(spec)) in
      if not(andred(snd(getCons(post, benv, spec)),
                 fn(nc) => Value.valuebool(beval(nc,
                                                 Env.append(newenv, benv),
                                                 spec))))
        then raise Postcondition_not_satisfied
        else newenv end end
and
```

```
    funcall(name, args, env, spec) =
  if isAbsFunction(name, len args, spec) then
    let val absfun = getAbsFunction(name, spec) in
      Env.lookup(geneval(primb true, getABody(absfun),
                         TEnv.extend(getAArgs(absfun), name,
                                     getARetType(absfun)),
                         primi 0::args, [name], env, spec),
                 unbound(name))
    end
  else let val friendfun = getFriend(name, args, spec) in
      Env.lookup(geneval(getPre(friendfun), getPost(friendfun),
                         TEnv.extend(getArgs(friendfun), "result",
                                     getRetType(friendfun)),
                         primi 0::args,
                         "result"::getModifies(friendfun),
                         env, spec),
                 unbound("result"))
  end
and
    methcall(obj, name, args, env, spec) =
  let val idofobj = if Env.isbound(env, bound(getident(obj)))
                       then bound(getident(obj))
                       else unbound(getident(obj))
      val typ = Env.getType(env, idofobj)
      val oper = getOp(name, getClass(typ, spec)) in
    Env.lookup(geneval(getPre(oper), getPost(oper),
                       TEnv.extend(TEnv.extend(getArgs(oper), "result",
                                               getRetType(oper)),
                                   "self", typ),
                       obj::primi 0::args,
                       "result"::getModifies(oper), env, spec),
               unbound("result"))
  end
and
     eval(object, opname, args, env, specification) =
  let val typ = if isfriend(object) then voidtype else
                Env.getType(env, bound(object))
      val oper =
        if isfriend(object) then getFriend(opname, args, specification)
        else getOp(opname, getClass(typ, specification))
      val modslist = if getRetType(oper) = voidtype then getModifies(oper)
                     else "result"::getModifies(oper)
 in
    makeNewEnv(env, object,
               geneval(getPre(oper), getPost(oper),
                       fixForms(getArgs(oper), object, typ, getRetType(oper)),
                       fixArgs(args, object, typ, getRetType(oper)),
                       modslist, env, specification),
               getArgs(oper), args, getModifies(oper))
  end;

fun declare(name, typ, env) =
      Env.extend(env, bound(name), Value.undef, typ);
```

B An Example SPECS-C++ Specification

This appendix contains the abstract (SML) syntax for the example presented in the paper.

```
val te = [
  ("OrderedPair", tupletype [("first", inttype), ("second", inttype)]),
  ("Relation", tupletype [("theRel", settype(typename("OrderedPair")))])];

val opc = {
  name = "OrderedPair",
  parentclass = none,
  invariant = primb true,
  absfunctions = [],
  protectedops = [],
  publicops = [
   operation {
    name = "OrderedPair",
    args = [("f", inttype), ("s", inttype)],
    returntype = voidtype,
    modifies = ["self"],
    pre = primb true,
    post = andexp(eq(field("first", primed("self")), ident("f")),
                  eq(field("second", primed("self")), ident("s")))}]};

val relc = {
  name = "Relation",
  parentclass = none,
  invariant = primb true,
  absfunctions = [],
  protectedops = [],
  publicops = [
   operation {
    name = "Relation",
    args = [],
    returntype = voidtype,
    modifies = ["self"],
    pre = primb true,
    post = eq(field("theRel", primed("self")), buildset [])},

   operation {
    name = "Insert",
    args = [("elem", typename("OrderedPair"))],
    returntype = voidtype,
    modifies = ["self"],
    pre = primb true,
    post = eq(field("theRel", primed("self")),
              union(field("theRel", ident("self")),
                    buildset [ident("elem")]))},

   operation {
    name = "RelTo",
    args = [("key", inttype)],
    returntype = settype(inttype),
    modifies = [],
    pre = primb true,
    post = andexp(forall("x", field("theRel", ident("self")),
                         eq(field("first", ident("x")), ident("key")),
                         member(field("second", ident("x")), result)),
                  forall("x", result, primb true,
                         exists("p", field("theRel", ident("self")),
                                eq(ident("x"), field("second", ident("p")))))
                         )}
 ]};

val s = {classes = [opc, relc], typeenv = te};
```

C SML Test Code

This appendix contains SML code for testing the specification presented in the previous appendix.

```
val e1 = declare("r", typename("Relation"), Env.empty);

val e1 = eval("r", "Relation", [], e1, s);

val e1 = declare("e", typename("OrderedPair"), e1);

val e1 = eval("e", "OrderedPair", [primi 1, primi 2], e1, s);

val e1 = eval("r", "Insert", [ident("e")], e1, s);

val e1 = eval("e", "OrderedPair", [primi 2, primi 2], e1, s);

val e1 = eval("r", "Insert", [ident("e")], e1, s);

val e1 = eval("e", "OrderedPair", [primi 2, primi 3], e1, s);

val e1 = eval("r", "Insert", [ident("e")], e1, s);

Value.tkString(Env.lookup(e1, bound "r"));

val e1 = eval("r", "RelTo", [primi 2], e1, s);
```

D Output

This appendix contains the output produced by running the test code presented in the previous appendix.

```
Standard ML of New Jersey, Version 0.93, February 15, 1993
Beta release version
val it = () : unit
- [opening prog.ml]
...
- [opening rep.ml]
...
- [opening test.ml]
val e1 = [(bound "r",undefined,typename "Relation")] : Env.E

val e1 = [(bound "r",Tuple [(#,#)],typename "Relation")] : Env.E
val e1 =
  [(bound "e",undefined,typename "OrderedPair"),
   (bound "r",Tuple [(#,#)],typename "Relation")] : Env.E

val e1 =
  [(bound "e",Tuple [(#,#),(#,#)],typename "OrderedPair"),
   (bound "r",Tuple [(#,#)],typename "Relation")] : Env.E

val e1 =
  [(bound "e",Tuple [(#,#),(#,#)],typename "OrderedPair"),
   (bound "r",Tuple [(#,#)],typename "Relation")] : Env.E

val e1 =
  [(bound "e",Tuple [(#,#),(#,#)],typename "OrderedPair"),
   (bound "r",Tuple [(#,#)],typename "Relation")] : Env.E

val e1 =
  [(bound "e",Tuple [(#,#),(#,#)],typename "OrderedPair"),
   (bound "r",Tuple [(#,#)],typename "Relation")] : Env.E

val e1 =
  [(bound "e",Tuple [(#,#),(#,#)],typename "OrderedPair"),
   (bound "r",Tuple [(#,#)],typename "Relation")] : Env.E

val e1 =
  [(bound "e",Tuple [(#,#),(#,#)],typename "OrderedPair"),
   (bound "r",Tuple [(#,#)],typename "Relation")] : Env.E
val it = "({(1, 2), (2, 2), (2, 3)})" : string
{2, 3}
val e1 =
  [(bound "e",Tuple [(#,#),(#,#)],typename "OrderedPair"),
   (bound "r",Tuple [(#,#)],typename "Relation")] : Env.E
val it = () : unit
-
```