Executing Formal Specifications with Constraint Programming

Tim Wahls
Iowa State University

Gary T. Leavens
Iowa State University

Albert L. Baker
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Programming Languages and Compilers Commons, and the Theory and Algorithms Commons

Recommended Citation
Computer Science Technical Reports. 124.
http://lib.dr.iastate.edu/cs_techreports/124

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Executing Formal Specifications with Constraint Programming

Abstract
We have implemented a technique for execution of formal, model-based specifications. The specifications we can execute are written at a level of abstraction that has not previously been supported in executable specification languages. The specification abstractions supported by our execution technique include quantified assertions that reference post-state values, and indirect definitions of post-state values (definitions that do not use equality). Our approach is based on translating specifications to the concurrent constraint programming language AKL. While there are, of course, expressible assertions that are not executable, our technique is amenable to any formal specification language based on a finite number of intrinsic types and pre- and postcondition assertions.

Keywords
Executable specifications, constraint programming, SPECS-C++ AKL, behavioral interface specification, C++, model-based specification, precondition, postcondition, formal specification, prototyping, constraint solving, constraint satisfaction, sets, unions, sequences, tuples, objects, first-order predicate calculus

Disciplines
Programming Languages and Compilers | Theory and Algorithms

Comments
Executing Formal Specifications with Constraint Programming

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

TR #97-12b

An earlier version of this report was titled “Executing Formal Specifications with Constraint Satisfaction.”

**Keywords:** Executable specifications, constraint programming, SPECS-C++ AKL, behavioral interface specification, C++, model-based specification, precondition, postcondition, formal specification, prototyping, constraint solving, constraint satisfaction, sets, unions, sequences, tuples, objects, first-order predicate calculus.


Submitted for publication.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA
Executing Formal Specifications with Constraint Programming

Tim Wahls*
wahls@psu.edu
Department of Mathematical and Computer Sciences, Penn State Harrisburg,
777 W. Harrisburg Pike, Middletown, PA 17057 USA

Gary T. Leavens†
leavens@cs.iastate.edu
Department of Computer Science, 229 Atanasoff Hall,
Iowa State University, Ames, Iowa 50011-1040 USA

Albert L. Baker‡
baker@cs.iastate.edu
Department of Computer Science, 228 Atanasoff Hall,
Iowa State University, Ames, Iowa 50011-1040 USA

May 2, 2000

Abstract
We have implemented a technique for execution of formal, model-based specifications. The specifications we can execute are written at a level of abstraction that has not previously been supported in executable specification languages. The specification abstractions supported by our execution technique include quantified assertions that reference post-state values, and indirect definitions of post-state values (definitions that do not use equality). Our approach is based on translating specifications to the concurrent constraint programming language AKL. While there are, of course, expressible assertions that are not executable, our technique is amenable to any formal specification language based on a finite number of intrinsic types and pre- and postcondition assertions.

1 Introduction
Executable specifications [AEeL92] [TY92] [Hen86] [TC89] [HH88] [O’N92a] [KK93] have several advantages over nonexecutable specifications — as tools for validating specifications against informal requirements, for prototyping, and for testing that implementations satisfy specifications. However, as Hayes and Jones have pointed out [HJ89], executable specifications are usually inferior to nonexecutable ones in a number of important ways, including:

- executable specifications typically suffer from implementation bias and overspecification. A specification that includes an algorithm implementing the specified functionality can influence developers to use that algorithm in an implementation. However, the algorithm used in a specification is rarely the best algorithm for an implementation. If developers do use different data structures and algorithms in the implementation, it is easier to formally verify that the implementation satisfies a more abstract specification than to verify against a biased specification that uses different data structures and algorithms.

---

*Wahls's work was supported in part by a fellowship from IBM Rochester.
†Leavens's work was supported in part by the National Science Foundation under grants CCR-9593108 and CCR-9803843.
‡Baker's work was supported in part by the National Science Foundation under grant CCR-9803843.
• Nonexecutable specifications are often nondeterministic or underspecified, either because a range of behaviors are permissible or because the specified system is nondeterministic. However, executable specifications are almost always deterministic.

• Nonexecutable specifications typically use features of first order logic such as negation, conjunction in which the order of conjuncts is not significant, disjunction (often resulting in nondeterminism), and quantification. Use of these features greatly increases the expressiveness of specifications, but complicates execution.

Previously, a developer who wished to use formal specifications had to make a choice between executable and nonexecutable specifications, and either choice sacrificed many of the advantages of the other. The execution technique we describe in this paper offers a third choice. Our execution technique uses the syntax of a nonexecutable specification language and can execute specifications written at nearly the same level of abstraction as typical nonexecutable specifications. In particular, our technique does not require the specifier to provide algorithms, can execute nondeterministic specifications cleanly, and can execute many specifications that use negation, quantifiers, disjunction, and so on. Hence, our execution technique alleviates the problems of executable specification languages to a larger degree than any other that the authors are aware of, and so does not force the specifier to give up the advantages of nonexecutable specifications to gain executability. Our system is publicly available for research use [Wah00].

The specification language that we execute is SPECS-C++ [WLB97], a model-based formal specification language designed for specifying the interfaces of C++ [Str91] classes. This language was not designed with executability in mind, and in fact our approach cannot execute all SPECS-C++ specifications. Our approach makes little use of the C++ specific features of SPECS-C++ except as described in Section 3, and so should be suitable for executing specifications written in other model-based specification languages such as VDM [Jon90] [A+93], Z [Hay93] [Sp98] [Sp92] and JML [LBR98] with only straightforward modifications.

We execute SPECS-C++ specifications by translating them to Agents Kernel Language (AKL) [JH94], and then executing the resulting AKL program. AKL is described in Section 2, so here we simply note that we selected AKL as a translation target after evaluating a number of other possibilities, including Prolog [CM84] [Col85] [Coh85], clp(FD) [Dia94], and DFKI Oz [M+98]. AKL works well as a translation target largely because it has a simple syntax and because it supports constraint programming cleanly. While our implementation depends on AKL, our ideas do not and would work with other languages with similar capabilities.

The rest of this paper elaborates the idea of executing formal specifications by translating them to constraint programs. In Section 2 we provide a brief introduction to AKL and constraint programming. In Section 3 we discuss model-based specification languages and the features of such languages that complicate their translation to constraint programs. Section 4 provides some example SPECS-C++ specifications and comments on executing these examples. In Section 5, we describe our technique for translating specifications into AKL by giving a (partial) denotational semantics for the translation algorithm and sketching a proof of the soundness of this algorithm. We also characterize the run time performance of the AKL programs thus generated. Section 6 outlines future work, and Section 7 briefly describes related work and summarizes our contributions.

2 Agents Kernel Language (AKL)

AKL [JH94] is based on concurrent constraint programming. A computation state consists of a set of concurrent agents that operate on a shared set of constraints called a constraint store. A constraint is an occurrence of one of the built-in relations (=, ≠, < etc.) with its actual arguments. For example, the following is a (comma-separated) list of constraints:

\[ x \equiv 3, y < 1.7, z = [a, b, c] \]

As in Prolog, logical variables begin with upper case letters, and square brackets are used to construct lists. The inequality relation is written \(\equiv\).
An agent can perform two operations on the constraint store. It can add a constraint to the
store (called *telling* the constraint), or it can check if a constraint is entailed (implied) or disentailed
by the store (called *asking* the constraint). If telling a constraint makes the store inconsistent, then
the computation fails. If an asked constraint is neither entailed nor disentailed, then the agent will
typically wait until sufficient constraints are added (by other agents) so that the asked constraint is
either entailed or disentailed.

An agent is usually written as a sequence of *clauses*. For example, the following agent `plus`
consists of three clauses (one per line in this case). This agent implements an addition relation that
computes any one of its arguments when the other two are known. Each clause contains an optional
*guard* and a *body*. Both the guard and the body consist of a conjunction of constraints and calls to
agents. Roughly speaking, when the constraints in the guard (or implied by the calls in the guard)
are asked, if they are all entailed, then the constraints in the body (or implied by the calls in the
body) of that clause are told. In each of the three clauses for agent `plus`, the guards are to the left
of the `|` symbols, and the bodies are to the right. When the guard of one of the clauses becomes
entailed, the body of that clause is evaluated and any resulting constraints are told. If more than
one guard is entailed, then the choice of which to use is (theoretically) nondeterministic for this
guard operator (the `|` called the *commit* operator in AKL). If all guards are disentailed, then the
agent fails. The agent `number` is built-in to AKL. If its argument is bound, `number` is true if that
argument is a number and false otherwise. If its argument is unbound, then `number` suspends and
waits for its argument to become bound (by other agents telling constraints).

```prolog
plus(X, Y, Z) :- number(X), number(Y) | Z is X + Y.
plus(X, Y, Z) :- number(X), number(Z) | Y is Z - X.
plus(X, Y, Z) :- number(Y), number(Z) | X is Z - Y.
```

Some of the features that make AKL useful as a translation target for specifications are:

- the order of conjuncts is relatively unimportant. For example, the following query:

```
| ?- X < 3, X = 2.
```

will fail when presented to a Prolog interpreter, because the variable X is not bound when the
comparison is evaluated. However, when this same query is presented to the AKL interpreter,
the constraint `X < 3` is told, and then the constraint `X = 2` is told. Since these constraints are
consistent, the solution `X = 2` is found and printed.

This is important for executing specifications because the order of conjuncts is *not* significant
in specifications. That is, the (partial) specification:

```
x' < 3 \land x' = 2
```

is equivalent to:

```
x' = 2 \land x' < 3
```

and from the specifier’s viewpoint, there is no reason why one should be executable and the
other not. Note that the constraint `X = 2` can act either to supply a value for `X`, or to check the
consistency of any value that `X` already has. Because the order that constraints are processed
in is not significant, the order of the conjuncts that the constraints are generated from is not
significant.

- constraints are *propagated* efficiently. Any addition to the constraint store can potentially allow
  many suspended agents to wake, execute and tell additional constraints. Since the effect of a
  constraint is to limit the range of values that a variable may have and still be consistent with
  the constraint store (the variable’s *domain*), telling a single constraint can potentially restrict
  the domains of many variables through constraint propagation.
An alternative view of this process is to think of all possible bindings for the variables in an AKL program as forming a search tree of possible outputs from the program. The leaves of the search tree have a binding for all variables, and interior nodes have bindings for only some of the variables. Each constraint in the constraint store makes some branches of the tree inconsistent with the store, and so the leaves of such branches are not valid outputs from the program. Constraint propagation has the effect of pruning this tree (by adding more constraints) and so helps in finding valid outputs more quickly by eliminating invalid outputs from consideration.

In terms of executing specifications, use of disjunctions, existential quantification and certain built-in operations of standard specification languages (for example, set membership and subset) in specifications leads to branches in the search tree for the AKL program that the specification is translated to. Hence, effective constraint propagation is critical for executing such specifications in a reasonable amount of time.

- alternative solutions can be obtained through backtracking. If multiple branches of the solution tree are consistent with the constraint store after an AKL program executes, the leaves of these branches can often be visited and displayed by backtracking. This corresponds to systematically enumerating the results of making different choices at all points in the execution where a choice was made.

A specification that is underspecified or nondeterministic is translated to an AKL program that has such choice points. Hence, backtracking allows one to determine whether or not the specification was underspecified, and if so, to enumerate multiple solutions that satisfy the specification.

- finite domain constraints can be used to make constraint propagation more efficient. In AKL, the domain of a variable can explicitly be set to be a finite subset of the integers. For such a finite domain variable, finite domain constraints can be used to restrict the domain efficiently. For example, the domain of the variable \( X \) can be specified in AKL as the integers 1 through 10. If the finite domain constraint \( X < 5 \) is then told, then the domain of \( X \) is immediately restricted to 1 through 4. This allows much more efficient pruning of the search tree than simply telling the constraint \( X < 5 \), because much of some subtree might be explored via backtracking before an ordinary constraint from the constraint store is checked for consistency with the solution associated with that subtree. The subtree for a solution that is inconsistent with the domain of a finite domain is never explored. Many constraint languages (including CHIP [Van88], clp(FD) [Dia94], and DFKI Oz [M*88]) use a similar notion of finite domains, and the theory is well established.

In our translation, an existentially quantified variable that ranges over some subset of the integers in a specification becomes a finite domain variable in the generated AKL program. We describe this process in more detail in Section 5.

3 Model-based Specification Languages

Model-based specification languages describe the behavior of programming language functions and procedures in terms of an underlying mathematical model. The most prominent examples are VDM [Jon90] [A+93] and Z [Hay93] [Sp89] [Sp92], although many other model-based specification languages have been developed. The model typically includes primitive types such as integers and characters, as well as more structured types such as finite sets, sequences, tuples, and functions. Constants, functions and relations on the model types provide the vocabulary for specifying procedures via first-order predicate logic pre- and postconditions. These conditions describe, respectively, sufficient conditions for the specified procedure to execute correctly, and if these are satisfied, what is guaranteed to be true when the procedure terminates. In other words, the precondition describes what must be true of the prestate (the program state just before the body of the specified procedure is run) for the procedure body to execute correctly, and the postcondition describes the post-state
resulting from the execution of the procedure. Hence, for our purposes, execution of such specifications means checking that the pre-state satisfies the precondition, and then finding post-state values that satisfy the postcondition.

The execution technique described in Section 5 was developed in the context of the specification language SPECS-C++ [WLBB87], which is a formal and model-based language specialized for specifying the interfaces of C++ classes [Str91]. However, the execution technique is not limited to SPECS-C++, as it depends only on first order assertions over a fixed set of model types, as found in all model-based specification languages. In fact, the addition of C++ specific features only complicates the execution of specifications. These features include classes, objects, and inheritance.

As we will be presenting the execution technique and sketching a proof of its soundness using denotational semantics [Sch86] techniques, we give the syntax for executable SPECS-C++ class specifications using syntax domains and abstract syntax as is typical for denotational semantics. (We do not have space for the full semantics and proof. See [WL00] for a fuller presentation.) Figure 1 gives the syntax domains. The domain Identifier represents variable, parameter, type and function names. The domain Literal represents the primitive values of SPECS-C++ (integer, float, double, character and string). Since these domains are well-known they are not further defined in the syntax. The remaining domains will all be defined syntactically.

Figure 2 gives BNF rules defining the syntax domains. The domains section is for defining types used in the specification. The data members section defines the components of instances of this class — i.e., an instance of the class can be represented as a tuple of the data members. The constraints section gives any invariant associated with instances of this class. The abstract functions section (domain F) defines a list of abstract functions, which are like the specification functions of VDM [Jon90] [A+93], and so are not part of the client interface of the class specification. The public: section (domain M) defines a list of post-style operation specifications with C++ interfaces. Domain B includes the standard first order boolean operations (including quantifiers \forall and \exists), and domain E gives all operations on expressions. Template class specifications and the specifications of the protected and private interfaces of a class are not yet executable, and so are not included. SPECS-C++ operators are discussed in the following text, and the structure of assertions is discussed in Section 5. We present several examples of SPECS-C++ specifications in Section 4.

The built-in types of SPECS-C++ are defined as follows. We use Object in the sense of location or l-value, which allows us to specify mutation and aliasing. Type Instance represents instances of classes. We model instances as a location (containing a tuple composed of the values of the data members of the class\(^1\)) and an identifier containing the name of the class that the instance belongs to. In our algorithm (and semantics), we only need to know the class of an instance when we have the location of that instance. Type Value is a disjoint union of all of the built-in types. Type

\(^1\)Tuples in SPECS-C++ are similar to records in other languages. In particular, the fields of a SPECS-C++ tuple are named, and so each data member is represented by a field with the same name as the data member.
C ::= class I {  /* model  
   domains  
   D  
   data members  
   A  
   constraints  
   B  
   abstract functions  
   F  */  
   public:  
   M  
   };  
S  
D ::= D_1;D_2 | T I | \epsilon  
T ::= char | int | float | double | bool | string | T & | set of T |  
   sequence of T | tuple(P)  
A ::= A_1;A_2 | T I | \epsilon  
F ::= F_1;F_2 | define I(P) as T such that B | \epsilon  
P ::= P_1;P_2 | T I | \epsilon  
M ::= M_1;M_2 | \epsilon  
   \* pre: B_1  
   \* modifies: L  
   \* post: B_2 */  
Y ::= void | T | \epsilon  
L ::= E | E, L | \epsilon  
B ::= true | false | !B | B \& B | B \& B | B => B  
   \forall T I [(I \in E \& B) => B] | \exists T I [B] |  
   E_1 = E_2 | E_1 != E_2 | E_1 < E_2 | E_1 <= E_2 | E_1 > E_2 | E_1 >= E_2 |  
   E_1 \in E_2 | E_1 \subset E_2  
E ::= I | V | first(E) | header(E) | last(E) | trailer(E) | length(E) |  
   domain(E) | range(E) | E_1 + E_2 | E_1 - E_2 | E_1 * E_2 | E_1 / E_2 | E_1 \% E_2 |  
   E_1 \union E_2 | E_1 \intersection E_2 | E_1 || E_2 | \{L\} | <L> | \langle L \rangle | \q E |  
   \q (E \& E) | \{I : I \in E \& B\} | B  
S ::= S_1;S_2 | T I | I_1.I_2(L) | I = E | I_1 = I_2.I_3(L)  

Figure 2: BNF rules defining the syntax domains.
Storable-value includes these value types and also logical variables, as discussed in Section 5.1.

\[
\begin{align*}
\text{Char} &= \text{Integer} \\
\text{Float} &= \text{Real} \\
\text{Double} &= \text{Real} \\
\text{String} &= \text{List(Storable-value)} \\
\text{Set} &= \text{List(Storable-value)} \\
\text{Sequence} &= \text{List(Storable-value)} \\
\text{Tuple} &= \text{Identifier} \rightarrow \text{Storable-value} \\
\text{Object} &= \text{Location} \\
\text{Instance} &= \text{Location \times Identifier}
\end{align*}
\]

\[
\begin{align*}
\text{Value} &= \text{Char} + \text{Integer} + \text{Float} + \text{Double} + \text{Boolean} + \text{String} + \text{Set} + \text{Sequence} \\
&\quad + \text{Tuple} + \text{Object} + \text{Instance} \\
\text{Storable-value} &= \text{Value} + \text{Variable}
\end{align*}
\]

The operation on these types are defined informally as follows. The pre-state value of an object is extracted by the postfix function \(\_\), and the post-state value by the postfix function \(\.'\). Thus, \(x\_\) is the pre-state value of \(x\), and \(x\.'\) is its post-state value. Note that we allow objects to be contained within values (for example, a set or sequence of objects, a tuple with a field of type object, or an object containing another object). This allows detection and specification of aliasing, which is critical for specifying the interfaces of C++ classes. Specification languages such as VDM and Z allow the specifier to distinguish between pre-state and post-state values, but do not have a complete and uniform theory of objects, and do not allow the detection or specification of aliasing.

In SPECS-C++, the only observer function on tuples besides equality \(=\) is the dot \(\_\) function — i.e. the extraction of a field from the tuple using the name of the field, as in \(T\_\text{count}\) for tuple \(T\) and field \(\text{count}\). Tuples are constructed by parenthesizing a list of values. The observers of sets are \(\text{\in}\) (set membership), \(\text{\subset}\), and \(\text{size}\) (cardinality). Sets are constructed by listing values in curly brackets \(\{\}\), by set comprehensions (the set of all values that satisfy some condition), and by the standard set operations, written \(\text{\cup}\), \(\text{\cap}\), and \(\text{\setminus}\) (for set difference). Besides the \(\text{length}\), \(\text{\in}\), and \(\text{\_based}\) indexing (using array-like \([\]\) notation) functions on sequences, SPECS-C++ defines additional sequence functions as follows, where \(<x\_1, \ldots, x\_n>\) is sequence constructor notation, and \(||\) is used for appending sequences:

\[
\begin{align*}
\text{first}(<a>||s) &= a \\
\text{last}(s||<c>) &= c \\
\text{header}(s||<c>) &= s \\
\text{trailer}(<a>||s) &= s
\end{align*}
\]

In terms of inheritance (and subtyping), our experience is that these relationships in most specifications of C++ classes are easily implemented by syntactic changes to the specification — for example, copying the specification of an operation from the specification of the base class to the specification of the derived class [DL96]. Hence, our execution technique assumes that such changes are made before the specification is executed, and so does not deal with inheritance and subtyping directly.

In terms of execution of SPECS-C++ (and other model-based specification languages) by translation to a constraint programming language, the key issues are translating specification syntax such as quantifiers and negation into executable constraint programs, and providing a library that implements the operators of SPECS-C++ and additional functionality needed for handling state transformations (pre-state to post-state) and universally quantified assertions. We return to these issues after presenting some examples of executable SPECS-C++ specifications.

4 Examples

In this section, we present three example specifications that our technique can execute. However, all of these specifications are typical (in our experience) of specifications written without executability
class List {
   /* model */
   domains
       sequence of int listtype;
   data members
       listtype tl;
   abstract functions
       define sorted(listtype l) as bool such that
           result = tobool(\forall int i [ 1 <= i < length(l) =>
           l[i] <= l[i + 1] ]);)
       define permutation(listtype l1, listtype l2) as bool such that
           result = tobool(range(l1) = range(l2));
   */
   public:
       List();
       /* modifies: self */
       post: tl' = <> */
       void insert(int i);
       /* modifies: self */
       post: tl' = <i> || tl' */
       void sort();
       /* modifies: self */
       post: permutation(tl', tl') \& sorted(tl') */
};

Figure 3: The specification of a list class. The SPECS-C++ keyword result is used to refer to the result of the function.

in mind.

Figure 3 gives a specification of a list class, including a constructor List for initializing instances of the class, and a function insert for adding an element to the beginning of a list. Function sort specifies a sorting operation. The range operator used in abstract function permutation returns a multiset of the elements of a sequence, so the call to permutation in the post-condition of sort correctly specifies that the post-state value is a permutation of the pre-state value. Since range returns a multiset, this specification can handle sequences with duplicates. The post-condition of sort also calls the abstract function sorted to check that the post-state value is sorted. Function sorted is passed the post-state value of the sequence object l, and so in fact the body of sorted is evaluated over a post-state value. This is significant because no other executable specification language that the authors are aware of would allow a post-state value to be referenced inside a universally quantified assertion in this way. Also note that this sort specification is not biased toward a particular sorting algorithm — in fact, this specification could easily have been written by a specifier working in a nonexecutable specification language. The price for this level of abstraction in the specification is that the execution is inefficient. The resulting AKL program will backtrack through permutations of the sequence (by running range backward) until a sorted permutation is found. It is also worth noting that this specification is underdetermined if the input sequence has
duplicates (because the duplicates can appear in different orders in the result). Someone executing the specification can observe this, because the AKL program will backtrack and present multiple outputs (even though they appear to be identical).

One concession to executability in the body of abstract functions sort and permutation is the use of tobool to mark an assertion used in a term context. In other words, this operator is used when the boolean value of an assertion is needed (to be returned or to specify a post-state value of an object holding a boolean value, for example). We introduced this operator because merging the term and assertion grammars introduced parsing conflicts that we were unable to resolve otherwise.

Figure 4 presents an interface specification of a C++ class Graph. Member function maxclique returns a clique of maximum size. It makes extensive use of abstract function cliquei, which takes a graph G, an integer i and a set of vertices vs, and returns true precisely when vs is a clique of size i in the graph G. During execution, the agent generated from this function is “run backward” by calling the agent with the vertex set argument vs unbound and letting the agent find a clique of the specified size and place it in vs if one exists. This is accomplished in the specification by passing an existentially quantified variable as the vertex set argument. “Running backward” in this manner is equivalent to “specification by inverse” [HJ89], and so is another way in which our technique can execute specifications that other techniques can not.

Function maxclique also shows a constructive use of negation in its last conjunct (used for finding a largest clique). This use of negation is similar to examples used by Hayes and Jones [HJ89] to argue that many specifications using negation are inherently nonexecutable. This example is executable because the possible values for the variable vs (the bound variable in the negated existentially quantified assertion) are restricted by function cliquei, and so the generated AKL program need not search an infinite domain for vs. However, this restriction is a necessary part of the specification, and not something added to permit execution. The specification of maxclique is again nondeterministic, and someone executing the specification can see all cliques of maximum size through backtracking. Execution of a call to maxclique is accomplished by checking all subsets of the set of vertices (see function cliquei) to find subsets that are maximal cliques, and so the running time is exponential in the number of vertices in the graph. (Since the problem is NP-hard, this is actually a reasonable algorithm.)

Figure 5 presents the specification of a C++ class Knapsack, which is intended to find solutions to an instance of the fractional knapsack problem. The basic idea is that a thief robbing a store can only carry a limited amount of goods in his knapsack, and wants to make sure that the value of these goods equals or exceeds some minimum value. In this instance, the store has 10 gold ingots, 20 silver ingots, and 25 brass ingots. Each gold ingot weighs 5 units and is worth $10, each silver ingot weighs 3 units and is worth $5, and each brass ingot weighs 2 units and is worth $1. Member function solve takes the capacity of the knapsack (in units) and the minimum value as parameters, and finds a choice of the number of ingots of each metal that the thief should take such that the total weight is at most the given capacity and the total value is at least the given minimum (if such a choice exists). In our implementation, this specification can also be used to find all such choices. As with our other examples, this specification has minimal implementation bias.

Because the variables (data members) in this specification are integers and are restricted to finite domains, the AKL code generated from this specification can take advantage of finite domain constraints. For example, once values for any two of gold, silver, and brass are fixed, the domain of the third variable is completely known and no additional search is needed to find all values for this third variable. In fact, depending of the values of the parameters passed to solve, some pruning of the domains of the three variables may be possible before any search is undertaken.

5 Executing Model-Based Specifications

At the top level, we execute SPECS-C++ specifications by the following process:

1. translate the SPECS-C++ specification and code for testing the specification (declarations of instances of the class and other variables, and calls to the member functions) to a syntax tree using standard compiler techniques (Flex [Pax90] and Bison [DS95]),
class Graph {
    /* model */
    domains
        int vertex;
        set of vertex vertexset;
        tuple (vertex v1, vertex v2) edge;
        set of edge edgeset;
    data members
        vertexset V;
        edgeset E;
    abstract functions
        define edgein(vertex v1, vertex v2, Graph G) as bool such that
            result = tobool((v1, v2) \in G.E \ (v2, v1) \in G.E);
        define cliquei(Graph G, int i, vertexset vs) as bool such that
            result = tobool(vs \subset G.V
                \ |vs| = i
                \forall vertex v \in vs =>
                    \forall vertex v2 [
                        v2 \in vs \forall v1 != v2 =>
                            edgein(v1, v2, G)]);}
} /* public:
    Graph(vertexset vs, edgeset es);
    /* modifies: self */
    post: V' = vs
        E' = es
    */
    vertexset maxclique();
    /* post: \exists int i \leq |V| 
        \forall \exists vertexset vs [ 
            cliquei(self^*, i, vs) 
            \ /
            \forall int j [i < j \leq |V| => 
                \not\exists vertexset vs2 [cliquei(self^*, j, vs2)]])
    */
};

Figure 4: The specification of a C++ class Graph. An (undirected) graph is modeled as a set of integer vertices V and a set of edges E consisting of pairs of vertices.
class Knapsack {
  /* model */
  data members
    int gold; int silver; int brass;
  */
  public:

    void solve(int capacity, int value);
  /* modifies: self */
  post: 0 <= gold' <= 10 \ 0 <= silver' <= 20
       \ 0 <= brass' <= 25
       \ gold' * 5 + silver' * 3 + brass' * 2 <= capacity
       \ gold' * 10 + silver' * 5 + brass' * 1 >= value
  */
};

Figure 5: The specification of a C++ class Knapsack, which represents an instance of the fractional knapsack problem.

2. typecheck the syntax tree and make some small modifications to ease code generation,
3. replace all variables bound by quantifiers with fresh variables to prevent capture of free variables during code generation,
4. generate an AKL program,
5. and finally, load the AKL program and the library of AKL agents used by generated programs into the AKL interpreter, and then execute the program. If the specification is consistent and executable, the user can also backtrack through the set of post-states that satisfy the specification, or ask for the entire set of post-states to be returned at one time. If the specification is inconsistent, the interpreter will return "no". Finally, if the specification is not executable, the interpreter will either go into an infinite loop or return "Computation suspended", which means that the program generated from the specification did not sufficiently define the post-state.

We describe this process in more detail by presenting a special-purpose denotational semantics that explains the execution technique. This formalizes the presentation and allows a proof of soundness. Our semantics has some unusual features designed to better model the execution technique. After the semantics and a sketch of the soundness proof, we provide a few notes on the implementation and end with a brief analysis of the running time of the generated AKL programs.

5.1 Denotational Semantics

Our basic approach in the semantics can be divided into two parts. First, we process a SPECS-C++ specification and translate it into a powerset of constraints, i.e., into a set of sets of constraints. The most important part of this translation is handling assertions (SPECS-C++ pre/post-conditions). We use a powerset of constraints because SPECS-C++ specifications can be nondeterministic (and also underdetermined), and so each simple set of constraints represents a different way of satisfying the assertion. Second, we take each set of constraints and and iteratively simplify it, which corresponds to a fixpoint construction in the semantics. This simplification can expose further nondeterminism, and so simplifying a set of constraints produces a set of results. Each result is a pair
Equal = Storable-value × Storable-value
NotEqual = Storable-value × Storable-value
Less = Storable-value × Storable-value
Member = Storable-value × Storable-value
Subset = Storable-value × Storable-value
Not = Boolean-expr × Environment × Store × Store
Forall = Identifier × Storable-value × Boolean-expr × Environment × Store × Store
First = Storable-value × Storable-value
Plus = Storable-value × Storable-value × Storable-value
Post = Storable-value × Storable-value
Call = Identifier × List(Storable-value) × Storable-value × Store × Store
False = Unit
:
Constraint = Equal + NotEqual + Member + Subset + Not + Forall First + Plus
+ Minus + Union + Post + Pre + Call + False + . . .

Figure 6: Type Constraint representing the constraints generated and simplified by the execution technique. Only a representative subset of the various types of constraints are presented.

consisting of an environment and a final store. Essentially, the fixpoint construction plays the role in the semantics that the AKL interpreter does in the implementation. This approach allows us to encapsulate the use of constraints in the semantics — the input to the semantics is a SPECS-C++ specification, and the output is a set of environment and store pairs. Hence, we can reason about the soundness of the algorithm without reference to a particular constraint programming language such as AKL.

An alternative approach to this semantics is for the valuation functions to return constraint logic programs, rather than sets of final states directly. The advantages of this approach are that it would allow us to take advantage of existing formal semantics for such languages [NF89] [JMMS98], and that it would closely model the way our algorithm works. However, formal semantics for CLP languages tend to focus on issues of control built into the language (the order that rules are used in, the order that answers are returned in, etc.) that are not critical here. Additionally, such a semantics would not be a helpful way to explain our work for those unfamiliar with logic and CLP languages. Finally, our semantics allow us to reason about sets of final states directly. Existing semantics for logic and CLP languages tend to be either operational or continuation passing style denotational semantics, and so are more suited to reasoning about one result at a time.

In showing soundness for our algorithm, we concentrate on the execution of a single call to a SPECS-C++ function, and show that every result store from this semantics satisfies the specification of that function. In particular, we show that any store constructed as specified by the semantics satisfies the post-condition of the specification when used as the post-state store.

In the first part of our semantics, we translate a specification to a powerset of constraints. We introduce a domain Constraint that includes constraints for all of the built-in operators of SPECS-C++.

For operators that are not relations, the matching constraint is converted to a relation by making the last “parameter” (the last element of the cartesian product) the result of the operation. This is the standard technique for converting arbitrary functions to relations in logic programming. The constraint False is used as a constraint that is always false. This is needed for explicitly indicating a failure in constraint simplification. We also define constraints representing universal quantification (Forall) and negation (Not). Figure 6 gives the definition of a representative subset of domain Constraint. As type Storable-value includes variables, the values stored in constraints can
$C$: Class-specification $\rightarrow$ Environment $\rightarrow$ Class-Environment $\rightarrow$

$\text{PowerSet(}\text{Environment } \times \text{Store}\text{)}$

```c
C[\text{class I}}$

/* model

domains

D
data members

A
constraints

B
abstract functions

F
*/

public:

M
}

S[] = \lambda e.\lambda ce. \text{let } e' = F[F] e \text{ in }

(\lambda (d', c, s). \text{if } c = \{} \text{ then } \{(e', s)\} \text{ else } \{\})^+

S[S][e', updatece I (M[M] e' \lambda i.\bot) ce, \{],

newstore, newnextv)
```

Figure 7: Function $C$ for evaluating a class specification.

be (or contain) variables.

Figure 7 presents $C$, the valuation function for class specifications. The Environment and Store parameters are standard, except that identifiers/locations are allowed to map to logical variables as well as values. In fact, environments can hold function definitions as well, giving:

$\text{Function} = \text{List(}\text{Storable-value}) \rightarrow \text{PowerSet(}\text{Constraint}) \rightarrow \text{Store} \rightarrow \text{Store} \rightarrow$

$\text{Storable-value} \rightarrow \text{NextVar} \rightarrow$

$\text{PowerSet(}\text{PowerSet(}\text{Constraint}) \times \text{Store} \times \text{NextVar}\text{ })$

$\text{Denotable-value} = \text{Function} + \text{Value} + \text{Variable}$

We compile abstract and member functions to type $\text{Function}$, store the result in an environment as appropriate, and then use these functions to generate constraints for the body of an abstract or member function when called.

The $\text{Class-Environment}$ is a mapping from each class name to an environment containing the member functions of that class, and so will be used for retrieving the proper member function for the instance used in a call. The denotations of the abstract functions are added to the environment by valuation function $F$. The denotations of the member functions are stored in a new environment by $M$ (see Figure 8 for both $F$ and $M$), which is then associated with the class name in the class environment. Both environments are used by $S$ to evaluate the “test case” (sequence of declarations, member function calls, and assignment statements) for the specification. $S$ returns a set of three-tuples (final environment, constraint set, final store) representing the final state. A set of states is necessary because specifications are often underdetermined or nondeterministic. If the constraint set in a state is empty, then the associated environment and store form a valid final state (answer) for the specification — they satisfy the specification and the test cases given as input to $C$. If the constraint set is not empty, the environment and store may or may not satisfy the specification and test cases, and so do not contribute to the final state. A nonempty constraint set results from the incompleteness of the execution technique — the technique was not strong enough to simplify the
\[ F : \text{Abstract-function} \rightarrow \text{Environment} \rightarrow \text{Environment} \]
\[ F[F_1; F_2] = \lambda e. F[F_1] \{ F[F_1] \} e \]
\[ F[\text{define } I(P) \text{ as } T \text{ such that } B = \lambda e. \]
\[ \text{let } f = \lambda \lambda e. s_{\text{pre}}, s_{\text{post}}, \lambda \text{nextv}. (\lambda (e', \text{nextv}). \{(e', s_{\text{post}}, \text{nextv})\})^+ \]
\[ \mathcal{B}[B] (\mathcal{P}[P] l e v, c, \text{nextv}) s_{\text{pre}} s_{\text{post}} \text{ in} \]
\[ F[e] = \lambda e. c \]

\[ M : \text{Member-function} \rightarrow \text{Environment} \rightarrow \text{Class-Environment} \]
\[ M[M_1; M_2] = \lambda e ce. M[M_2] e (M[M_1] e ce) \]
\[ M \circ Y I(P); \]
\[ \text{/* pre: } B_1 \]
\[ \text{modifies: } L \]
\[ \text{post: } B_2 * / \]
\[ \text{let } f = \lambda \lambda e. s_{\text{pre}}, s_{\text{post}}, \lambda \text{nextv}. \]
\[ \text{let } e' = \text{update env} \{ \text{self} \} \text{ hd}(l) (\mathcal{P}[P] \text{ tl}(l) e v) \text{ in} \]
\[ (\lambda (e'', s'', c'', \text{nextv''}). \{(e'', s'', \text{nextv''})\})^+ \]
\[ ((\text{fix } R)^+) ((\lambda (l, c', \text{nextv})). \]
\[ \text{let } (s'_{\text{post}}, \text{nextv'}) = \text{makefresh}(s_{\text{pre}}) \text{ lst nextv' in} \]
\[ (\text{adjustState } e' s_{\text{pre}} s'_{\text{post}} \]
\[ (\lambda (e'', \text{nextv''}). \mathcal{B}[B_2] \]
\[ (e', c'', \text{nextv''}) s_{\text{pre}} s'_{\text{post})^+ \]
\[ \mathcal{B}[B_1] (e', c', \text{nextv'}) s_{\text{pre}} s_{\text{pre}} \) \text{ in} \]
\[ \text{update env} \{ I \} \text{ inFunction}(f) ce \]
\[ M[e] = \lambda e ce. ce \]

Figure 8: Function \( F \) for storing abstract function definitions in an environment, function \( M \) for storing member function definitions, and function \( P \) (not shown, see [WL00]) that sets up the environment to evaluate the function body in.

remaining constraints.

To extend functions to deal with set-valued inputs, we use the notation \( f^+ \). Given a function \( A : T_1 \rightarrow \text{PowerSet}(T_2), A^+ : \text{PowerSet}(T_1) \rightarrow \text{PowerSet}(T_2) \) is defined as: \( A^+(X) = \bigcup \{ A(x) \mid x \in X \} \).

Figure 8 presents \( F \) and \( M \), the valuation functions for abstract and member functions, respectively. In both cases, the function is compiled to type \( \text{Function} \), and then associated with its name in an environment so that it can be invoked later. Abstract functions are placed in the top-level (global) environment, while the member functions of each class are stored in a class environment. Abstract functions can call other abstract functions (and themselves recursively). We allow recursion by including the name of an abstract function in a \( \text{Call} \) constraint (see Figure 10) and then retrieving the function from the environment when simplifying such a constraint (see Figure 14). This delay ensures that all abstract functions are stored in the environment before any call to an abstract function is evaluated.

For member functions (valuation function \( M \)), some additional work is required. The \textit{modifies} clause specifies what objects can change from the pre-state to the post-state [GHG93] [Win87] [BMR95], and so this clause is used to construct the post-state store. The list of objects in the \textit{modifies} is used by function \textit{makefresh}, which returns the same store that it is passed, except that each object in the list is bound to a fresh variable. Note that the post-state store passed to a member
B: Boolean-expr → (Environment × PowerSet(Constraint) × NextVar) →
Store → PowerSet(PowerSet(Constraint) × NextVar)

\[ B[true] = \lambda (e, c, nextv). s_{pre}. s_{post}. \{ (e, nextv) \} \]

\[ B[false] = \lambda (e, c, nextv). s_{pre}. s_{post}. \{ \{ \text{inFalse}() \}, nextv \} \]

\[ B[B_1 \land B_2] = \lambda (e, c, nextv). s_{pre}. s_{post}. \{ (e \cup \{ \text{inNot}(B_1, e, s_{pre}, s_{post}) \}, nextv) \} \]

\[ B[B_1 \lor B_2] = \lambda (e, c, nextv). s_{pre}. s_{post}. \{ (\lambda(e', nextv'). (\lambda(e'', nextv''). \{ (e' \cup e'' \cup nextv'') \})^+) \}
\]

\[ B[B_1 \Rightarrow B_2] = \lambda (e, c, nextv). s_{pre}. s_{post}. \{ (\lambda(e', nextv'). (\lambda(e'', nextv''). \{ (e' \cup e'' \cup nextv'') \})^+) \}
\]

\[ B[\forall T \forall I \{ I \in E \land B_1 \Rightarrow B_2 \}] = \lambda (e, c, nextv). s_{pre}. s_{post}. \{ \lambda(e', nextv'). (\lambda(e'', nextv''). \{ (e' \cup e'' \cup nextv'') \})^+ \}
\]

\[ B[\exists T \exists I \{ B \}] = \lambda (e, c, nextv). s_{pre}. s_{post}. \{ \lambda(e', nextv'). (\lambda(e'', nextv''). \{ (e' \cup e'' \cup nextv'') \})^+ \}
\]

\[ B[E_1 = E_2] = \lambda (e, c, nextv). s_{pre}. s_{post}. \{ (e' = e'' \cup \{ \text{inEqual}(v_1, v_2) \}, nextv'') \}^+
\]

\[ B[updateEnv \{ \} v e, c, nextv'] = \lambda (e, c, nextv). s_{pre}. s_{post}. \{ (e' = e'' \cup \{ \text{inEqual}(v_1, v_2) \}, nextv'') \}^+
\]

end

Figure 9: Function B for evaluating boolean assertions into sets of constraints.

function is ignored — this store is only passed in because it is needed for abstract functions. After constraints are generated from both the pre- and postcondition, functional R (see Figure 11) is used in a fixpoint construction to simplify the constraints into an environment and result store. The fixpoint construction is needed because this simplification is iterative, and should continue until no more constraints can be simplified. Functional R is used for member functions because they are called from outside of the class — abstract functions are only called from member functions and other abstract functions, and so the constraints generated from such calls will be simplified with the other constraints generated from the body of the member function. Function adjustState (see Figure 12) is used to put the arguments into the right form for applying R.

Valuation function S (not shown, but described in [WL00]) provides the semantics for syntax domain Statement — the “test cases” (where each test case is a sequence of object declarations, calls to member functions, and assignment statements) used for executing class specifications. For declarations of objects (including instances that are objects), the object name is associated with a newly-allocated location in the environment. For member function calls, the class of the object receiving the message is retrieved and then passed to the class environment to get the environment containing the (compiled versions of the) member functions for that class. The name of the called function is then used to obtain the member function from that environment. The member function is then called with the appropriate arguments.
Figure 9 presents valuation function $B$, which is used to translate assertions (type Boolean-expr) into constraints. $B$ makes use of $E$ (Figure 10), which translates expressions (terms) to constraints. $B$ and $E$ both use function $newvar$ to generate fresh variables. Parameter $nextv$ of type $NextVar$ is essentially a counter used to ensure that each call to $newvar$ returns a variable with a distinct name. In $B$, negated assertions are converted directly to constraints to be evaluated later. Note that the environment and stores are included so the assertion can be evaluated in the proper context. For conjunctions, powersets of constraints are collected from both conjuncts. The powerset of constraints returned contains one set for each way of choosing a set from each of these powersets. For disjunctions, each disjunct is translated and the resulting powersets of constraints are unioned. We consider a powerset of constraints to be consistent if any element of the powerset is a consistent set of constraints, so this gives the correct semantics. An implication $P \Rightarrow Q$ is translated as the equivalent $(P \lor Q) \land \neg P$.

 Universally quantified assertions can only be executed if the domain that the bound variable ranges over is known and finite. The variable’s domain can be given either by restricting it to be an element of a set or sequence, or by bounding it as a finite subrange of the integers (the syntax for a finite subrange is not shown, but is simply $E < I < E$, where $<$ can be used in place of either or both $<=$). The domain is evaluated, and included in a $Forall$ constraint to be evaluated later. Existential quantification simply introduces a fresh variable.

 If an expression of the form: $E < I < E$ (where again $<$ can be used in place of either or both $<=$ and $I$ is an integer variable) is not used to bound a universally quantified variable, then $I$ is translated as a finite domain variable in the implementation (this is not shown in the semantics), using the domain explicitly given. This translation is correct, as semantically the effect of using a finite domain variable is to constrain the variable to be one of the values given in its domain. Currently, this is the only situation in which we are using finite domain variables, and so this is a major area for future work.

 The remaining boolean assertions are applications of relations, and are translated directly to the matching constraints. The case for relation $=$ is shown. The remaining cases (including cases for relations \in and \subset) are similar.

 Function $E$ (Figure 10) translates expressions into sets of constraints. Since there is a matching constraint for each built-in operator of SPECS-C++, this consists of translating the actual operands into sets of constraints, and then building the appropriate constraint. In the majority of cases, this is simple enough to essentially be a type conversion — we are taking an element of domain Expression and converting it to an element of domain Constraint. However, we do lose some structure in this process. For example, an arithmetic expression like $x * 2 + y$ is translated (roughly) to: $\{inTimes(x, 2, v), inPlus(v, y, v')\}$, where $v$ and $v'$ are fresh variables and $v'$ represents the overall result. This flattening of the structure of the expression implies that only local constraint propagation is done — no propagation is done based directly on the larger surrounding context of a term. However, this does accurately model the current version of the translation algorithm.

 Many of the cases for $E$ are similar, and in fact most are omitted from Figure 10. However, the case for elements of syntax domain Boolean-expr (for $B \in$ Boolean-expr) is considerably different. This case is for elements of Boolean-expr used in an expression context. Hence, we must evaluate an element of Boolean-expr for its value, rather than just generating constraints from it. We divide this case into two subcases. If we already know what truth value $B$ must have for the set of constraints to be consistent, we simply generate constraints for $B$ or $\neg B$ as appropriate. If the needed truth value for $B$ is unknown, then we return the union of two sets of states. For one set, we generate constraints for $B$ and specify that the value of $B$ is true. For the other set, we generate constraints for $\neg B$ and specify that the value of $B$ is false. If the specification determines the value of $B$, all states in one of these sets must fail and so not contribute any answers to the final result.

 Once the specification has been translated into a powerset of constraints, the functional $R$ (Figure 11) is used to simplify each set of constraints into a set of stores. $R$ is always used in a fixed point construction, as constraint simplification proceeds until no more simplification is possible — which occurs when an attempt at simplification does not change the set of constraints. $R$ makes heavy use of function $S$ which attempts to simplify a single constraint. Several of the more interesting cases of function $S$ are presented in Figures 13 and 14. For each kind of constraint, $S$ determines if
\[ E : \text{Expression} \rightarrow \text{Storable-value} \rightarrow \]
\[ \text{(Environment} \times \text{PowerSet(Constraint)} \times \text{Next Var}) \rightarrow \text{Store} \rightarrow \text{Store} \rightarrow \]
\[ \text{PowerSet(Constraint)} \times \text{Next Var} \]
\[ E[B] = \lambda v.\lambda (e, c, nextv).\lambda s_{pre}.\lambda s_{post}. \]
\[ \text{cases } v \text{ of} \]
\[ \quad \text{isValue}(b) \rightarrow \]
\[ \quad \quad \text{if } b \text{ then } B[B] (e, c, nextv) s_{pre} s_{post} \]
\[ \quad \quad \text{else } B[B] (e, c, nextv) s_{pre} s_{post} \]
\[ \quad \text{let } (v', nextv') = \text{newvar}(nextv) \text{ in} \]
\[ \quad (\lambda (c', nextv'). \left\{ (e' \cup \{\text{in\text{Prev}}(v', v)\}, nextv') \right\})^+ \]
\[ \text{E}[E[v']] = \lambda v.\lambda (e, c, nextv).\lambda s_{pre}.\lambda s_{post}. \]
\[ \text{let } (v_1, nextv') = \text{newvar}(nextv) \text{ in} \]
\[ \text{let } (v_2, nextv'') = \text{newvar}(nextv) \text{ in} \]
\[ (\lambda (c', nextv''). \left\{ (e'' \cup \{\text{in\text{Prev}}(v_1, v_2, v)\}, nextv'') \right\})^+ \]
\[ \text{E}[E[v_1 + E_2]] = \lambda v.\lambda (e, c, nextv).\lambda s_{pre}.\lambda s_{post}. \]
\[ \text{let } (v_1, nextv') = \text{newvar}(nextv) \text{ in} \]
\[ \text{let } (v_2, nextv'') = \text{newvar}(nextv) \text{ in} \]
\[ (\lambda (c', nextv''). \left\{ (e'' \cup \{\text{in\text{Prev}}(v_1, v_2, v)\}, nextv'') \right\})^+ \]
\[ \text{E}[E[v_1]] = \lambda v.\lambda (e, c, nextv).\lambda s_{pre}.\lambda s_{post}. \]
\[ \quad (\lambda (v, c', nextv'). \left\{ (e' \cup \{\text{in\text{Prev}}(v, v, v)\}, nextv') \right\})^+ \]
\[ \text{E}[E[l]] (e, c, nextv) s_{pre} s_{post} \]
\[ \vdots \]
\[ \text{end} \]

Figure 10: Representative cases of valuation function \( E \).

\[
R : \text{(Environment} \times \text{Store} \times \text{Store} \times \text{PowerSet(Constraint)} \times \text{Next Var}) \rightarrow \]
\[ \text{PowerSet(Constraint)} \times \text{Store} \times \text{Store} \times \text{PowerSet(Constraint)} \times \text{Next Var} \]
\[ R = \lambda v.\lambda (e, s_{pre}, s_{post}, cs, nextv). \text{if } cs = \{ \text{ then } \left\{ (e, s_{pre}, s_{post}, cs, nextv) \right\} \]
\[ \text{else } \bigcup \left\{ (s \in s_{post}, cs \Rightarrow \{ e \}, \text{nextv}) \mid e \in cs \right\} \]

Figure 11: The functional \( R \) used for simplifying constraints. The least fixed point of \( R^+ \) is the denotation of an assertion. \( R \) uses function \( S \) (defined in Figures 13 and 14) to simplify each constraint.
adjustState : Environment → Store → Store →
(PowerSet(PowerSet(Constraint)) × NextVar) →
PowerSet(Environment × Store × Store × PowerSet(Constraint) × NextVar)
adjustState = λe.λspre.λspost.λcs. {\{e, spre, spost, c, nextv\} | (c, nextv) ∈ cs}

Figure 12: Function adjustState, which is used to adjust the representation of a state to the form needed for applying R.

enough information is available to make a simplification step. If not, then S leaves the constraint set unchanged (by returning the constraint to the set). Otherwise, S performs a simplification, in which case the constraint used is not returned to the constraint set. However, simplifying a constraint may cause one or more new constraints to be added, and could also cause values in the current post-state store to become more defined. This models the constraint propagation used in the implementation. We use unify to mean the standard unification algorithm, except that this version returns a set of constraints rather than a substitution, using the obvious equivalence between a substitution and a set of equality (Equal) constraints. We use the notation [v'/v] X to denote replacing all occurrences of v by v' in X, where v must be a variable, v' is a Variable or a Value, and X is an environment, store or constraint set. Note that elements of Variable can occur within each of these structures.

In the following, we comment on the specific simplification done for several of the kinds of constraints.

For Forall constraints, S first determines if the domain of the bound variable is known. If so, then several new constraints are added to the constraint set: one for choosing some element v of the domain, one for producing a new domain v' which is the original with v removed, and one new Forall constraint that uses the new domain v'. Finally, constraints are collected from the body of the assertion, with the bound variable associated with v in the environment.

For Not constraints, S first evaluates the assertion contained in the constraint. If all states in this result (referred to as res in S) have empty constraint sets, then the contained assertion is true, and so the Not constraint and thus the current constraint set are unsatisfiable. This is indicated by replacing the current constraint set with {inFalse()}. If all constraint sets of the states in res contain inFalse(), then the contained assertion is false, and so the Not constraint succeeds. Otherwise, S cannot determine the satisfiability of the Not constraint.

False constraints are unsatisfiable, and so cause the current constraint set to be replaced by {inFalse()}. Equal constraints are handled by a combination of substitution and use of the standard unification algorithm. Other kinds of constraints are handled as described earlier.

5.2 Proof of Soundness

In this section, we sketch a proof of the soundness of valuation functions B (Figure 9) and \( \mathcal{E} \) (Figure 10), and of the soundness of \((fix \ R^+)(\) (Figure 11). These are the parts of the semantics that deal directly with assertions, and so we can show soundness by showing that any store produced by the semantics satisfies the assertion that was input. Omitted parts and details of the proof are contained in [WL00].

The valuation functions that precede B (C, M, F and S in Figures 7 and 8) describe translating larger parts of SPECS-C++ specifications. To show soundness for these functions, we would need to show that they are correct with respect to some independent formal semantics for SPECS-C++. While useful, this part of the proof is not as interesting as showing soundness for assertions, and so we omit it.

We state the soundness result for assertions as a theorem below. The notation B[e, spre, spost] means the truth value of B under the given environment and stores.
\[ S : \text{Constraint} \rightarrow \]
\[ \langle \text{Environment} \times \text{Store} \times \text{Store} \times \text{PowerSet}(\text{Constraint}) \times \text{NextVar} \rangle \rightarrow \]
\[ \text{PowerSet}(\text{Environment} \times \text{Store} \times \text{Store} \times \text{PowerSet}(\text{Constraint}) \times \text{NextVar} ) \]
\[ S = \lambda e. \lambda (e, s_{pre}, s_{post}, cs, nextv). \text{cases } cs \text{ of} \]
\[ \text{cases } ds \text{ of} \]
\[ \text{isValue}(d) \rightarrow \text{cases } ds \text{ of} \]
\[ \text{isSet}(vl) \rightarrow (\text{if } vl = \text{nil} \text{ then } \{(e, s_{pre}, s_{post}, cs, nextv)\}) \]
\[ \text{else let } (v, nextv') = \text{newvar}(nextv) \text{ in} \]
\[ \text{let } e'' = \text{updateenv } [I] v e' \text{ in} \]
\[ \text{let } (e'', nextv'') = \text{newvar}(nextv') \text{ in} \]
\[ (\lambda(cs', nextv''). \{(e, s_{pre}, s_{post}, cs' \cup \text{inMember}(v, ds), \text{inMinus}(ds, v, e'), \text{inForall}(I, e''', B[e'''], s_{pre}', s_{post}')}, \text{nextv''})\})^+ \]
\[ B[B] (e'', cs, nextv'') s_{pre}' s_{post} \]
\[ \text{else } \{(e, s_{pre}, s_{post}, \text{inFalse}(), nextv)\} \]
\[ \text{end} \]
\[ \parallel \text{isVariable}(i) \rightarrow \{(e, s_{pre}, s_{post}, cs \cup} \]
\[ \text{inForall}(I, ds, B[e'''], s_{pre}', s_{post}')], \text{nextv})\} \]
\[ \text{end} \]
\[ \parallel \text{isNot}(B, e'', s_{pre}', s_{post}) \rightarrow \]
\[ \text{let } res = \left(\text{fix } R^+\right) (\text{adjustState } e s_{pre} s_{post} (B[B] (e'', cs, nextv) s_{pre}' s_{post}')) \]
\[ \text{in if } \forall (e'', s_{pre}'', s_{post}'', cs', nextv') \in res . \text{cs'} = \{\} \text{ then} \]
\[ \{(e, s_{pre}, s_{post}, \text{inFalse}()), \text{nextv}\} \]
\[ \text{else if } \forall (e'', s_{pre}'', s_{post}'', cs', nextv') \in res . \text{inFalse} \in \text{cs}' \text{ then} \]
\[ \{(e, s_{pre}, s_{post}, cs, nextv')\} \]
\[ \text{else } \{(e, s_{pre}, s_{post}, cs \cup \text{inNot}(B, e'', s_{pre}'', s_{post}''), \text{nextv})\} \]
\[ \parallel \text{isFalse}() \rightarrow \{(e, s_{pre}, s_{post}, \text{inFalse}(), nextv)\} \]
\[ \parallel \text{isEqual}(v_1, v_2) \rightarrow \text{cases } v_1 \text{ of} \]
\[ \text{isVariable}(i) \rightarrow \{(v_1 / v_1) e, s_{pre}[v_2/v_1] s_{post}[v_2/v_1] cs, nextv)\} \]
\[ \parallel \text{isValue}(n) \rightarrow \text{cases } v_2 \text{ of} \]
\[ \text{isVariable}(i') \rightarrow \{(v_1 / v_2) e, s_{pre}[v_1/v_2] s_{post}[v_1/v_2] cs, nextv)\} \]
\[ \parallel \text{isValue}(n') \rightarrow \text{let } (cs', nextv') = \text{unify}(v_1, v_2, cs, nextv) \text{ in} \]
\[ \{(e, s_{pre}, s_{post}, cs', nextv')\} \]
\[ \text{end} \]
\[ \text{end} \]

Figure 13: Representative cases of function S, which defines the effect of simplifying a single constraint. (Part 1 of 2.)
isFirst(v1, v2) → cases v1 of
  isVariable(i) → cases v2 of
    isVariable(i') → { (ε, s_pre, s_post, cs ∪ {isFirst(v1, v2)}, nextv) }
    □ isValue(ε) → let (nv, nextv') = newvar(nextv) in
      { (ε, s_pre, s_post, cs ∪
          {inConcat(inValue(inSequence(cons(v2, nil))), nv, v1)},
          nextv') }
  end
  □ isValue(seq) → cases seq of
    isSequence(l) → { (ε, s_pre, s_post, cs ∪ {inEqual(v2, hd(l))), nextv) }
    else { (ε, s_pre, s_post, [inFalse()], nextv) }
  end
  □ isPost(v1, v2) →
    cases v1 of
      isVariable(i1) → { (ε, s_pre, s_post, cs ∪ {inPost(v1, v2)}, nextv) }
      □ isValue(ε) → cases l of
        isObject(l') →
          let (cs', nextv') = unify(access l' s_post, v2, cs, nextv) in
          { (ε, s_pre, s_post, cs', nextv') }
        instance(l', ci) →
          let (cs', nextv') = unify(access l' s_post, v2, cs, nextv) in
          { (ε, s_pre, s_post, cs', nextv') }
        else { (ε, s_pre, s_post, [inFalse()], nextv) }
      end
    end
  □ isCall(I, lv, v, s_pre', s_post') →
    cases accessenv[I] e of
      isFunction(f) →
        (M(cs', s_post', nextv'). { (ε, s_pre, s_post, cs', nextv') } +
        (f cons(v, lv) cs s_pre' s_post' v nextv)
      else { (ε, s_pre, s_post, [inFalse()], nextv) }
  end
end

Figure 14: More representative cases of function S, which defines the effect of simplifying a single constraint. (Part 2 of 2.)
Theorem 5.1 Soundness Theorem:

\[ \forall B \in \text{Boolean-expr}. \forall (cs, \text{next} v') \in B[B][e, \{\}, \text{next} v) s_{\text{pre}}, s_{\text{post}}. \]
\[ \forall (e', s'_{\text{pre}}, s'_{\text{post}}, cs', \text{next} v'') \in (\text{fix } R^+) \{(e, s_{\text{pre}}, s_{\text{post}}, cs, \text{next} v')\}. \]
\[ cs' = \{\} \Rightarrow B[e, s_{\text{pre}}, s'_{\text{post}}] \]

That is, if the constraints derived from B are simplified into the environment \(e'\), pair of stores \(s'_{\text{pre}}\) and \(s'_{\text{post}}\), and set of constraints \(cs'\), and all constraints can be simplified (\(cs'\) is empty), then the assertion B is true under \(e, s_{\text{pre}}\) and \(s_{\text{post}}\). We ignore \(s'_{\text{pre}}\) and \(e'\) because simple inspection of the semantics shows that the pre-state store is never changed, and solving constraints never changes the environment. The environment and pre-state store are part of the result simply to make the type of \(R\) work out for the fixed point construction.

In the proof, we work with the following (stronger) formulation of this theorem to make the proof easier. In particular, we need an equivalence between the truth value of the assertion and the consistency of the constraint set generated from the assertion for proving the soundness of \(S\) for \(Not\) constraints. Following the standard definition, we define a set of constraints to be consistent if and only if there exists at least one labeling (assignment of values) of the logical variables of the constraint set that satisfies every constraint in that set under a given environment and pair of stores. This version of the theorem states that consistency of the set of constraints returned by the algorithm (under the environment and stores returned by the algorithm) is equivalent to the truth value of the original assertion under the same environment and stores. If the set of constraints is empty, then consistent is defined to be true. Hence, this second version of the theorem implies the original.

Lemma 5.2 Soundness Theorem (stronger version):

\[ \forall B \in \text{Boolean-expr}. \forall (cs, \text{next} v') \in B[B][e, \{\}, \text{next} v) s_{\text{pre}}, s_{\text{post}}. \]
\[ \forall (e', s'_{\text{pre}}, s'_{\text{post}}, cs', \text{next} v'') \in (\text{fix } R^+) \{(e, s_{\text{pre}}, s_{\text{post}}, cs, \text{next} v')\}. \]
\[ \text{consistent}(cs', e, s_{\text{pre}}, s'_{\text{post}}) \Rightarrow B[e, s_{\text{pre}}, s'_{\text{post}}] \]

Note that we are not proving completeness (even though we are using \(\Leftrightarrow\)), because we do not show that \(\text{fix } R^+\) always simplifies a set of constraints to the empty set. In fact, \(\text{fix } R^+\) does not always simplify a set of constraints, even when such simplification is possible.

To show the Soundness Theorem, we use the following lemmas. First, we must show that the valuation function \(B\) is sound.

Lemma 5.3 Soundness of \(B\):

\[ \forall B \in \text{Boolean-expr}. \forall (cs, \text{next} v') \in B[B][e, \{\}, \text{next} v) s_{\text{pre}}, s_{\text{post}}. \]
\[ \forall (e', s'_{\text{pre}}, s'_{\text{post}}, cs', \text{next} v'') \in (\text{fix } R^+) \{(e, s_{\text{pre}}, s_{\text{post}}, cs, \text{next} v')\}. \]
\[ \text{consistent}(cs, e, s_{\text{pre}}, s'_{\text{post}}) \Rightarrow B[e, s_{\text{pre}}, s'_{\text{post}}] \]

That is, that under a valid final post-state store \(s'_{\text{post}}\), the set of constraints returned by \(B\) is consistent if and only if the assertion B is true.

Then we show that the construction (\(\text{fix } R^+\)) used for simplifying constraints is sound:

Lemma 5.4 Soundness of (\(\text{fix } R^+\)):

\[ \forall B \in \text{Boolean-expr}. \forall (cs, \text{next} v') \in B[B][e, \{\}, \text{next} v) s_{\text{pre}}, s_{\text{post}}. \]
\[ \forall (e', s'_{\text{pre}}, s'_{\text{post}}, cs', \text{next} v'') \in (\text{fix } R^+) \{(e, s_{\text{pre}}, s_{\text{post}}, cs, \text{next} v')\}. \]
\[ \text{consistent}(cs, e, s_{\text{pre}}, s'_{\text{post}}) \Leftrightarrow \text{consistent}(cs', e, s_{\text{pre}}, s'_{\text{post}}) \]

Together, Lemmas 5.3 and 5.4 imply the Soundness Theorem (via Lemma 5.2). While space precludes providing full proofs of these lemmas, we note that Lemma 5.3 is proved by induction on the structure of Boolean-expr and the definition of \(B\) given in Figure 9. To prove Lemma 5.4, we note that \(R\) simply applies function \(S\) repeatedly (in the fixed point construction) to simplify a set of
constraints (see Figure 11). Hence, we prove Lemma 5.4 by induction on the number of applications of $S$.

As part of the proof of Lemma 5.2, we must also prove the soundness of $E$. Although we again omit the proof, we state the relevant soundness result as a lemma.

**Lemma 5.5 Soundness of $E$:**

$$\forall E \in \text{Expression}. \forall (\text{cs}, \text{nextv'}) \in E\{E\} \ \text{vl} (e, \{\}, \text{nextv}) \ o_p r \ o_{p} \text{st}. \ \forall (e', o_{p}pr, o_{p}st, \text{nextv}) \in \{\text{fix } R^2\} \ \{(e, o_{p}pr, o_{p}st, \text{cs}, \text{nextv'})\}. \ \text{consistent}(e', e, o_{p}pr, o_{p}st) \ \Leftrightarrow \ \text{vl} = E[e, o_{p}pr, o_{p}st]$$

Note that the argument $\text{vl}$ to $E$ represents the value of the expression. Here, we mean $\text{vl}$ as a $\text{Value}$, not a $\text{Variable}$.

### 5.3 Notes on the Implementation

Like Prolog, AKL uses commas to separate conjuncts. Thus, translating a conjunction is straightforward. For a disjunction of the form $P \lor Q$, we generate a fresh agent name. This agent is later defined with two clauses — one corresponding to $P$ and the other to $Q$. In AKL, this agent will succeed if either clause succeeds, so this gives the correct semantics for disjunction. If both $P$ and $Q$ are true when this specification is executed, then both clauses for this new agent will succeed, and both possible sets of outputs can be visited through backtracking. We implement the generation of new agents later in the program by keeping a queue of agent definitions that need to be generated.

For a negated constraint $\neg P$, we generate a fresh agent name. This agent is later defined so that it will fail if $P$ is true, and succeed if $P$ is false. This is done by using the constraints generated from $P$ as the guard in a clause with the guard operator $\neg\neg$ (called $\text{then}$ in AKL), which requires that all guards in an agent except one be disentailed and the remaining guard entailed before the body of the clause with the entailed guard is executed. Hence, we can define an agent that succeeds precisely when the constraints generated from $P$ are disentailed as follows, where $\text{mp}$ is the agent name, each $a_i$ is a parameter, and $\text{CL}$ is the list of constraints generated from $P$:

$\text{mp}(a_1, a_2, \ldots a_n) :- \text{CL} -> \text{fail}.$

$\text{mp}(a_1, a_2, \ldots a_n) :- -> \text{true}.$

An implication $P \Rightarrow Q$ is transformed into code as if it were $(P \land Q) \lor \neg P$ (using the transformations already defined), which matches the semantics. The reason for using this approach rather than treating the implication as $\neg P \lor Q$ is that the evaluation of a guard is not allowed to modify the global constraint store, and so the usual interpretation of the implication would never allow constraints in the antecedent ($P$) to be told. This is problematic because it could result in legal outputs from the specification being missed. The interpretation used here is sound, as $(P \land Q) \lor \neg P$ is logically equivalent to $P \Rightarrow Q$.

Since variables that occur free in an AKL program are essentially existentially quantified, an existentially quantified assertion requires little special handling during translation. However, the translation must respect the scope of the quantified variable and ensure that the variable is not captured by a surrounding quantifier or a formal parameter when the quantifier is removed. This is implemented by renaming of all bound variables with new variable names that are guaranteed not to be used anywhere else in the program. Note that our approach does not require that the specifier supply an explicit domain for the existentially quantified variable. This allows existentially quantified assertions to be quite expressive — much more so than in other executable specification languages. For an example of this, see the use of existentially quantified assertions in the post-condition of $\text{maxclique}$ in Figure 4.

While the translation algorithm does not require that existentially quantified variables have an explicit domain, execution of the resulting code will fail unless the specification somehow defines a set of values for the variable. These limitations on the use of quantified assertions do make the executable subset of SPECS-C++ less powerful than the full language, but we have not found this limitation to be burdensome in practice. For example, a specification of a C++ class rarely uses
quantification over all of the integers. It is much more typical to quantify over the indices of a (finite) sequence, and this gives the bounds needed for execution. Our technique allows much more general use of quantified assertions than is available in any other executable specification language that we are aware of. Many other executable specification languages allow quantified assertions over finite domains to be executed, but almost all of these languages simply evaluate such assertions for their truth value, rather than using them directly in building post-state values. The fa3 system [KK33] is the only other execution technique that the authors are aware of that can use quantified assertions directly in defining the post-state, but fa3 can only use one restricted form of existential quantification in this way. The ability to execute quantified assertions over post-state values is one of the major advantages of our execution technique.

As previously noted, an assertion of the form: \( E \leq I \leq E \) where \( I \) is an integer variable not bound by a universal quantifier, is translated to a finite domain variable and the appropriate constraint for restricting its domain. All of the implementations of SPECS-C++ operations in the library use finite domain constraints when possible, so that an occurrence of \( < \) (for example) in a specification results in telling of a finite domain constraint if a finite domain variable is involved. Since such finite domain constraints are typically not sufficient to restrict the domain of a finite domain variable to one value, each finite domain variable is enumerated (forced to take each value remaining in its domain, in turn) in the generated code. Such enumeration is critical when finite domain variables are used, as otherwise a finite domain variable with multiple values remaining in its domain will cause execution of the generated code to suspend.

Both abstract (specification) and member functions in a SPECS-C++ specification are translated directly to an AKL agent with the same name. The formal parameters of the agent are the formals of the function, plus three additional parameters: the return value from the function (if needed), a pre-state store, and a post-state store. The store arguments are needed to capture any state changes specified by the abstract function. Stores are implemented as a list of object and value pairs. The body of the abstract function is an assertion that is translated as previously described.

The pre- and post-conditions of the member function specification are both assertions, and so both are translated as previously described. When the generated code is run, if the pre-condition is satisfied, then the code generated from the post-condition is executed. Otherwise, the error message is printed and the execution fails.

The library of AKL agents includes agents implementing all of the built-in operators of SPECS-C++ and some additional utility agents. The agents implementing SPECS-C++ operators propagate constraints by executing both in the normal way that the SPECS-C++ operators execute and also “backward” when necessary — i.e. computing the inputs of the SPECS-C++ operator given its output. One example of this is the library agent \texttt{plus} presented in Section 2. Another example from the agent library is the following agent for computing the length of a list (used in computing the size of a sequence or string). If the length of the list is known but nothing else is known about the list, then the first two clauses of this agent build a list of fresh AKL variables (the \_ character denotes a fresh variable in AKL). If the list is defined, then the second two clauses compute its length.

\begin{verbatim}
length(X, 0) :- | X = [].
length(X, N) :- N > 0 | minus(N, 1, N1), X = [\_ | X1], length(X1, N1).
length([], N) :- | N = 0.
length([\_ | XL], N) :- | length(XL, N1), plus(N1, 1, N).
\end{verbatim}

This technique increases constraint propagation because it allows the agent to execute (and add more constraints to the store) under a wider variety of conditions. We have found that agents are often executed “backward” when specifications are executed (see the use of the \texttt{range} agent in Figure 3 and of the \texttt{cliquei} agent in Figure 4).

Full source code for our system and working examples are publicly available for research use [Wah00].
The effect of constraint propagation on running time is difficult to quantify, and does not affect the worst case running time (although it can have a profound impact on the average case). Hence, we will simply give a way to bound the running time of generated AKL programs for the worst case, and then give running times for some of the specifications in Section 4. This time is determined by the size of the search tree explored during execution. The worst-case size of the search tree explored is \(O(a^n)\), where \(n\) is the number of variables in the problem and \(a\) is the maximum domain size for any of the variables. This analysis is precisely the same as that for standard constraint satisfaction problems [Ts93, p. 37]. In a SPECS-C++ postcondition, any post-state value is equivalent to a variable in a constraint satisfaction problem, and existentially quantifying over the \(m\) indices of a post-state sequence value introduces \(m\) additional variables. For example, in the following assertion:

\[
\exists \text{int } i \ [1 \leq i \leq 10 \land P(S'[i])]
\]

each \(S'[i]\) is treated as a separate variable constrained by the assertion \(P\).

Each value tried for such a variable is potentially a function of all of the quantified variables in the postcondition. Hence, the number of values that could be tried for each variable (i.e. the size of the domain of that variable) is bounded by the product of the sizes of the domains of the quantified variables. As an existential quantification over the indices of a post-state sequence value conceptually introduces more variables, the size of the domain of such a quantified variable does not contribute to this product. Suppose that a postcondition refers to \(i\) post-state values, and there are \(j\) existentially quantified variables over the indices of post-state sequence values, and the maximum domain size of such a quantified variable is \(n\). Additionally, suppose that there are \(k\) other quantified variables, and the maximum domain size quantified over for these variables is \(m\). Then, the worst case size of the tree searched is: \(O((km)^{i+j/n})\).

In practice, the ability to predict the time needed to execute a specification by inspecting the postcondition is useful because it provides an easy way to determine what size inputs (pre-state values) can reasonably be used. As a simple rule-of-thumb, a postcondition that does not contain an existential quantification over the indices of a post-state sequence value can usually be executed efficiently, and so it is practical to use large inputs. Otherwise, only small inputs can be used. However, even small “test cases” are helpful in debugging [JD96] and in demonstrating the functionality of specifications. Long execution times are a direct consequence of the level of abstraction of the specifications being executed.

We present some running times for the examples from Section 4 in Figure 15. These times were collected on a Sparc Ultra 10 running Solaris 7. For each run, the time (in seconds) to find and enumerate all solutions was measured. We give two running times each for \(\text{sort()}\) and \(\text{maxclique()}\) to show the effect of input size when the running time is exponential. The running time for \(\text{solve()}\) is also exponential (in the number of types of goods). These running times show that our system can execute interesting specifications in a reasonable amount of time.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Input Size</th>
<th># of Solutions</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{sort()}) in class List</td>
<td>5 (sequence length)</td>
<td>1</td>
<td>0.03</td>
</tr>
<tr>
<td>(\text{sort()}) in class List</td>
<td>8 (sequence length)</td>
<td>1</td>
<td>20.11</td>
</tr>
<tr>
<td>(\text{maxclique()}) in class Graph</td>
<td>6 vertices</td>
<td>2</td>
<td>0.34</td>
</tr>
<tr>
<td>(\text{maxclique()}) in class Graph</td>
<td>7 edges</td>
<td>2</td>
<td>7.45</td>
</tr>
<tr>
<td>(\text{solve()}) in class Knapsack</td>
<td>3 types of goods</td>
<td>70</td>
<td>0.92</td>
</tr>
</tbody>
</table>

Figure 15: Sample Running Times.
6 Future Work

One simple extension of our work is the execution of instantiations of generic specifications (in SPECS-C++, specifications of C++ template classes). Generic specifications are specifications parameterized by types, and instantiation of a generic specification occurs when an actual type is substituted for each parameter type. As the actual parameter type is known at execution time, the execution technique would need only minor changes to execute such specifications. Another simple extension would be to check that any instances of the class that are created or modified by a member function satisfy the invariant for the class (given by the constraints clause in the specification). Presently, the invariant is not used in executing a specification, but this kind of checking would be valuable in validating specifications.

More interesting future work centers on improving constraint propagation in the generated AKL programs, particularly with regard to finite domain variables and finite domain constraints. We would like to use finite domain constraints in more situations than is currently the case — perhaps by somehow translating domains of other types to integer domains. Additionally, the placement of finite domain constraints in an AKL program has a large effect on the efficiency of programs. For example, it is advantageous to delay enumerating a finite domain variable (forcing it to take each value of its restricted domain, in turn) as long as possible. This should be done by placing the constraint that forces enumeration as late in the generated program as possible. Our current translation strategy does not do enough analysis of the specification to determine the latest possible point where enumeration could be done. Improving placement of this constraint would greatly increase efficiency for some specifications.

Exploiting parallelism is another interesting possibility. The Penny language [MH97] [Mon97] is a parallel version of AKL that does not require the programmer to indicate parallelism explicitly. However, Penny is not source code compatible with AKL, and so using Penny would require retargeting the specification compiler and porting the agent library. We are still investigating the difficulty of this retargeting and evaluating the potential speed improvements to be gained by using Penny.

Another planned enhancement of our execution technique is a graphical user interface for constructing test cases and displaying the results. The idea is to provide a "drag-and-drop" environment for building pre-state values of the SPECS-C++ model types, a convenient interface for using these values as inputs to the generated AKL programs, and a nicely formatted display of the resulting post-state. A similar environment has already been developed [Gur00] for a previous version of our translator.

Another possible addition is support for calling the AKL code generated directly from a C++ program. This would let the specification serve as a prototype for the C++ class it specifies, and so would allow client code that used the class to be developed and tested before the class was implemented.

Finally, our technique is ideal for using specifications as test oracles, since it allows outputs from a specification and an implementation to be compared directly [GB94]. In fact, our technique can even test whether a deterministic implementation satisfies a nondeterministic specification in a much more efficient way than backtracking through all results from the specification and comparing them to the result from the implementation. A better approach is to run the implementation, and then run the specification with both the pre-state and the post-state known (from the result of running the implementation). When run this way, the specification will either succeed (if the given post-state is a possible result of the specification) or fail. Hence, we plan to implement an environment for using specifications as test oracles, perhaps as an extension of the graphical environment described above.

7 Related Work and Conclusion

In the literature on the execution of model-based specifications, three distinct approaches to executing specifications dominate:
1. Restrict the syntax of the language so that specifications can be directly executed. In particular, this usually means that references to post-state values cannot appear in the bodies of quantified assertions, and that only assertions of the form \( x' = \text{val} \) can be used to provide post-state values, where \( \text{val} \) is strictly an expression over pre-state values. This is the most prevalent approach, and examples include me too [Hen86], SMLVIEW [O’N92a] [O’N92b], the technique used for executing IPTES mini-specifications [ELA93] [LL91] [AEeL92], and EPROL [HI88] [HI86].

The \texttt{fasc3} execution technique [KK93] [Kra88] can execute a much larger set of assertions than any of the others mentioned here, but can not execute universally quantified assertions that refer to post-state values, and can only execute one very limited form of existentially quantified assertions that refer to post-state values. Our technique, however, can execute such assertions.

2. Use Prolog syntax or code in the specification language, and execute specifications via Prolog. Examples include PLEASE [TC89] and OBSERV [TY92]. This approach is problematic because Prolog syntax is quite limited when compared to the usual syntax of specification languages, and because some Prolog features have no logical interpretation and so should not be used in specifications.

3. Explicitly translate specifications to Prolog programs. Researchers have experimented with this approach [DKC90] [WE92], but to the authors’ knowledge, no completely automated translator exists.

Clearly, all of these approaches that are automated suffer from the criticisms leveled by Hayes and Jones [HJ89], as they either force overspecification (e.g., by requiring “extra” information used for execution as in Prolog programs), or greatly restrict the specification language constructs available to the specifier.

Although our approach requires the specifier to make some concessions to executability, it does execute specifications that are quite close to typical nonexecutable specifications. Our approach allows the execution of many specification techniques that are not executable by typical executable specification languages. These techniques include:

- use of quantified assertions with references to post-state values in their bodies
- nondeterministic/underdetermined specification
- nontrivial use of negation
- specification by inverse
- use of conjuncts that appear in any order

Section 4 contains examples of executable uses of each of these techniques. Making these specification techniques executable greatly increases the expressiveness of the resulting specifications. These specifications can serve the usual role of (nonexecutable) specifications (tools for documentation, reasoning, and verification) much more effectively than can other executable specifications. Hence, our approach allows specifiers to enjoy the benefits of using executable specifications without sacrificing many of the benefits of nonexecutable specifications.

References


