# Formal Semantics and Soundness of an Algorithm for Translating Model-based Specifications to Concurrent Constraint Programs

Tim Wahls and Gary T. Leavens

TR #00-02a
March 2000, Revised August 2000

**Keywords:** Executable specifications, behavioral interface specification language, formal specification, prototyping, translation of specifications to constraints, constraint programming, soundness, denotational semantics, model-based specification, formal methods, precondition, postcondition, SPECS-C++.

**1999 CR Categories:** D.2.1 [*Software Engineering*] Requirements/Specifications — Rapid prototyping, languages, theory, tools; D.2.4 [*Software Engineering*] Software/Program Verification — Formal methods, programming by contract, reliability, tools, SPECS-C++; D.2.7 [*Software Engineering*] Distribution, Maintenance, and Enhancement — Documentation; D.3.2 [*Programming Languages*] Language Classifications — Constraint and logic languages, nonprocedural languages, very high-level languages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, pre- and post-conditions, specification techniques; F.3.1 [*Logics and Meanings of Programs*] Semantics of Programming Languages — Denotational semantics.



The original version of this report was titled "Formal Semantics and Soundness of an Algorithm for Translating Model-based Specifications to Constraint Programs."

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

# Formal Semantics and Soundness of an Algorithm for Translating Model-based Specifications to Concurrent Constraint Programs

Tim Wahls*
Department of Mathematical and Computer Sciences
W256 Olmsted, Penn State Harrisburg,
777 W. Harrisburg Pike, Middletown, PA 17057 USA
wahls@psu.edu

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

August 24, 2000

**Abstract**

This paper presents an algorithm for executing formal specifications, and a proof of the soundness of that algorithm. The algorithm executes specifications written in the model-based specification language SPECS-C++ by transforming such specifications to concurrent constraint programs. This approach can execute specifications written at a high level of abstraction. Denotational semantics techniques are used for both explaining the algorithm and for proving its soundness.

## 1 Introduction

If a formal specification can be executed, then the specification can be validated against informal requirements and can be used as a prototype and test oracle. It is especially critical that any system for executing formal specifications be sound, as one important use of a formal specification is as the standard of correctness for an implementation. If a system for executing formal specifications is not sound, then specifications executed and tested using that system might contain undetected errors or the specification might be incorrectly modified to contain errors. Such errors are likely to be propagated to the implementation.

On the other hand, a formal specification should be written at a high level of abstraction — it should not suffer from implementation bias or overspecification, and should permit nondeterminism [HJ89]. But the farther removed the specification language is from a programming language, the more difficult it is to execute. Because the system for executing specifications is then larger and more complex, it is correspondingly more difficult to prove the soundness of that system.

This paper presents a proof of the soundness of an algorithm [WLB00] for translating SPECS-C++ [WBL94] specifications to programs in the concurrent constraint programming language AKL [JH94]. Our translation algorithm can execute specifications written at a very high level of abstraction, including specifications that use negation, nondeterminism, and quantified assertions that refer to post-state values. The proof is accomplished by giving a denotational semantics [Sch86] for the translation algorithm, and then proving the soundness of the valuation functions. The goals of this work are twofold: first, to present the translation algorithm in a relatively language independent manner, and second, to give confidence to those

*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 grant CCR-9803843.

using specification development and testing tools based on our algorithm that the answers returned are correct with respect to the specification.

The semantics given is designed for explaining and proving the soundness of the translation algorithm, and so is considerably different from a semantics designed only to present SPECS-C++. As SPECS-C++ uses first order predicate calculus assertions to specify the behavior of C++ [Str91] functions, a translation algorithm for SPECS-C++ is largely an algorithm for translating assertions. Soundness for such an algorithm means that the answers returned by the target program will satisfy the assertion that the program was generated from. However, the translation algorithm is not complete, so the program generated can fail to terminate even when the corresponding assertion is satisfiable.

## 1.1 Background

Before continuing with the semantics, we provide some brief background on SPECS-C++ and AKL.

SPECS-C++ is a model-based specification language, and so is similar to VDM [Jon90] [A+93] and Z [Hay93] [Spi89] [Spi92]. In particular, SPECS-C++ specifications consist of first order assertions written over a set of built-in types such as sets, sequences, tuples (records), C-style objects (in the sense of location or l-value) and primitive types such as integers and booleans. SPECS-C++ was designed for specifying the interfaces of C++ [Str91] classes, and in fact a SPECS-C++ specification can serve as a C++ header file, with the pre- and postconditions for the member functions given as C-style comments. However, SPECS-C++ was *not* designed to be executable, and in fact SPECS-C++ specifications are stylistically very similar to VDM and Z specifications. We present SPECS-C++ syntax and a simple example shortly.

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 ($=$, $\neq$, $<$ etc.) with its actual arguments. 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. While these features make AKL well suited for executing specifications, they complicate semantics and soundness proofs. The main problem is that agents can suspend, so the effect of an agent on the constraint store may not be apparent until some point well after the agent is first invoked.

## 1.2 Approach

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. We extend the built-in constraints of AKL with constraints for each relation and function of SPECS-C++. The most important part of this translation is handling assertions (SPECS-C++ pre- and postconditions). 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 consisting of an environment and a final store (a mapping from locations to values, not a constraint store). Essentially, the fixpoint construction plays the role in the semantics that the AKL interpreter does in the execution algorithm itself, with each set of constraints in the semantics representing a constraint store in the implementation. This approach allows us to model agents that can suspend, and also encapsulates 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 programming language such as AKL.

An alternative approach to this semantics is for the valuation functions to return concurrent constraint programs, rather than sets of final states directly. The advantages of this approach are that it would allow us the possibility of taking advantage of existing formal semantics for such languages [SRP91] (and possibly of logic and constraint logic programming languages [NF89] [JMMS98]), and that it would closely model the way our algorithm actually works. However, the language semantics just mentioned focus on formalizing the languages themselves, and so it is not always clear how well such semantics support proving the soundness of programs written in these languages. Additionally, such an approach would not be a helpful way to explain our work for those unfamiliar with concurrent constraint programming languages. Hence, we chose not to use this approach.

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 environment and store pair from this semantics satisfies the specification of that function. In particular, we show that if the precondition of the specification is satisfied by the pre-state store, then any store constructed as specified by this semantics satisfies the postcondition of the specification when used as the post-state store.

Although the semantics takes the form of a denotational semantics for SPECS-C++, our intent is to present the translation algorithm and provide a mechanism for proving its soundness. A semantics designed strictly for presenting SPECS-C++ would not use constraints, and so would be considerably simpler. Our implementation matches this semantics quite closely — the constraint stores produced by the implementation are isomorphic to the constraint sets produced by the valuation function in Section 4, and taking the fixpoint of function $R^+$ (also presented in Section 4) is an abstraction of the functionality of the AKL interpreter, together with the library of AKL agents used in the implementation. Hence, results proved for this semantics apply quite directly to our implementation.

### 1.3 Example

As an example of applying this semantics, consider the following specification for a simple SPECS-C++ function, where `x'` denotes the post-state value of object `x`.

```
void foo(int& x);
/* pre: true
   modifies: x
   post: x' \in {1, 2, 3} /\ x' != 2 */
```

This specification is nondeterministic, and so applying the first step of the semantics would yield three sets of constraints. These sets could be represented intuitively as follows (we use a more formal notation in presenting the semantics):

```
{x' = 1, x' != 2}
{x' = 2, x' != 2}
{x' = 3, x' != 2}
```

As each of these sets is simplified (in the second step of the semantics), the second set would be found to be inconsistent, and so would not contribute to the result. The remaining two sets would each yield one result, where the environments would be unchanged from the original in both cases, and the final store would bind the location of `x'` to 1 in the first case, and to 3 in the second. Both of these results are sound, as the specification of `foo` is satisfied if `x'` is 1, and also if `x'` is 3.

### 1.4 Overview

In the following sections, we give an abstract syntax for executable SPECS-C++ specifications, give the semantic algebras used, and define the valuation functions for the semantics. Finally, we show the soundness of (part of) the semantics by showing that if our semantics produces an answer for a call to a SPECS-C++ function, then that answer satisfies the specification of the function.

## 2 Abstract Syntax

In this section we explain the abstract syntax of an executable SPECS-C++ class specification. We refer to this syntax as abstract syntax for consistency with standard terminology in denotational semantics, Abstract syntax domains:

| | | |
|---|---|---|
| C ∈ Class-specification | F ∈ Abstract-function | S ∈ Statement |
| D ∈ Type-declaration | M ∈ Member-function | I ∈ Identifier |
| P ∈ Parameter | Y ∈ Result-type | B ∈ Boolean-expr |
| A ∈ Data-member | E ∈ Expression | V ∈ Literal |
| T ∈ Type-expression | L ∈ Expression-list | |

Figure 1: Syntax domains for the abstract syntax.

although the syntax presented is very close to the actual (concrete) syntax of SPECS-C++. As template class specifications and the specifications of the protected and private interfaces of a class are not yet executable, they are not included here.

Figure 1 gives the syntax domains for the abstract syntax. 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. We need very little of the semantics of types in the rest of the semantics, and so valuation functions are not given for domains Type-declaration, Data-member, Type-expression and Result-type. We use $\epsilon$ to denote an "empty" right hand side in a BNF rule.

Figure 2 gives BNF rules defining the syntax domains. Note that domain F defines a list of abstract functions, which are like the auxiliary functions of VDM [Jon90] [A+93]. M defines a list of pre-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.

# 3 Semantic Algebras

## 3.1 Values

The built-in types of SPECS-C++ are defined as follows. We use *Object* in the sense of location or l-value, and *Instance* for instances of classes. We model instances as a location (containing a tuple composed of the values of the data members of the class) and an identifier containing the name of the class that the instance belongs to. In our 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.

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

$$\begin{aligned}
Value = {} & Char + Integer + Float + Double + Boolean + String + Set + Sequence + Tuple + Object + \\
& Instance
\end{aligned}$$

Storable values include values of type *Value* as above, and also variables. We use variable here in the sense of logic programming, so that values can be unified. We allow a variable to appear anywhere a value can (in an environment or store, as part of another value ...) because specifications often give only partial

```
C ::= class I {
      /* model
        domains
          D
        data members
          A
        constraints
          B
        abstract functions
          F
      */
      public:
        M
      };
      S

D ::= D_1;D_2 | T I | ε
T ::= char | int | float | double | bool | string | T& | set of T | sequence of T | tuple(P)
A ::= A_1;A_2 | T I | ε
F ::= F_1;F_2 | define I(P) as T such that B | ε
P ::= P_1;P_2 | T I | ε
M ::= M_1;M_2 | ε |
      Y I(P);
      /* pre: B_1
        modifies: L
        post: B_2*/
Y ::= void | T | ε
L ::= L_1;L_2 | E | ε
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> | (L) | |E| | E.I | E' | E^ | E[E] | I(L)| result | self | {T 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.

information about post-state values – for example, values for only some index positions of a sequence or some fields of a tuple. Thus, we need a mechanism that allows us to keep track of the information about a value that is already known, and also that allows later constraints (involving the same variables) to produce more information about those values. Variables are modeled simply as integers — they will always be injected into another domain, so there is no danger of confusing variables with numeric values.

$Variable = Integer$
$Storable\text{-}value = Value + Variable$

### 3.2 Environments and Stores

As environments can also store the denotation of member and abstract functions, we give the type of that function denotation here. In type *Function*, the *List*(*Storable-value*) parameter represents the actual arguments to the function and the *Storable-value* parameter represents the function's result. Functions are treated as "constraint generators" — when called, they generate constraints as appropriate from their bodies. See Figure 5 for details of how SPECS-C++ functions are translated to this type. The use of type *NextVar* is explained below.

$$\begin{aligned} Function = {} & List(Storable\text{-}value) \to PowerSet(Constraint_\perp) \to Store \to Store \to Storable\text{-}value \to NextVar \\ & \to PowerSet(PowerSet(Constraint_\perp) \times Store \times NextVar) \end{aligned}$$

Denotable values are then just a disjoint union of functions and the storable values.

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

Now, environments (type *Environment*) can be defined simply as functions from identifiers to denotable values, as usual. Operations include *updateenv* to update the value associated with an identifier and *accessenv* to look up the value associated with an identifier.

Domain $Environment = \text{Identifier} \to Denotable\text{-}value_\perp$
Operations:
$updateenv : \text{Identifier} \to Denotable\text{-}value \to Environment \to Environment$
$updateenv = \lambda i.\lambda v.\lambda e.\ [i \mapsto v]e$
$accessenv : \text{Identifier} \to Environment \to Denotable\text{-}value_\perp$
$accessenv = \lambda i.\lambda e.\ (e\ i)$

We also define a class environment to associate a class name with the member functions of that class. This follows [LP98], except that we use the term class environment in place of method dictionary. For simplicity, we map a class name to an environment as defined above. This provides more generality than needed, as only function definitions will be stored in the environment associated with a class name. Operations *updatecenv* and *accesscenv* are analogous to the operations on environments mentioned above.

Domain $Class\text{-}Environment = \text{Identifier} \to Environment_\perp$
Operations:
$updatecenv : \text{Identifier} \to Environment \to Class\text{-}Environment \to Class\text{-}Environment$
$updatecenv = \lambda i.\lambda e.\lambda ce.\ [i \mapsto e]ce$
$accesscenv : \text{Identifier} \to Class\text{-}Environment \to Environment_\perp$
$accesscenv = \lambda i.\lambda ce.\ (ce\ i)$

The translation algorithm we are modeling is based on logic and constraint programming, and so makes heavy use of logical variables. In particular, we need the ability to generate fresh (unused) logical variables. We do this by threading a parameter of type *NextVar* (an integer counter) through the semantics and using it as an argument to function *newvar* to generate the "next" logical variable as a *Storable-value* value.

Domain $NextVar = Int$
Operations:

$Equal = Storable\text{-}value \times Storable\text{-}value$
$NotEqual = Storable\text{-}value \times Storable\text{-}value$
$Less = Storable\text{-}value \times Storable\text{-}value$
$Member = Storable\text{-}value \times Storable\text{-}value$
$Subset = Storable\text{-}value \times Storable\text{-}value$
$Not = \text{Boolean-expr} \times Environment \times Store \times Store$
$Forall = \text{Identifier} \times Storable\text{-}value \times \text{Boolean-expr} \times Environment \times Store \times Store$
$First = Storable\text{-}value \times Storable\text{-}value$
$Plus = Storable\text{-}value \times Storable\text{-}value \times Storable\text{-}value$
$Union = Storable\text{-}value \times Storable\text{-}value \times Storable\text{-}value$
$Post = Storable\text{-}value \times Storable\text{-}value$
$Comp = \text{Identifier} \times Storable\text{-}value \times \text{Boolean-expr} \times Storable\text{-}value \times Environment \times Store \times Store$
$Call = \text{Identifier} \times List(Storable\text{-}value) \times Storable\text{-}value \times Store \times Store$
$False = Unit$

$Constraint = Equal + NotEqual + Member + Subset + Not + Forall\ First + Plus + Minus + Union +$
$\quad Post + Pre + Comp + Call + False + \ldots$

Figure 3: Type *Constraint* representing the constraints generated and simplified by the execution technique. Only a representative subset of the various types of constraints is presented.

$newnextv\colon\ \to NextVar$
$\quad newnextv = 0$
$newvar : NextVar \to (Storable\text{-}value \times NextVar)$
$\quad newvar = \lambda n.\ (\text{in}Variable(n), n+1)$

The store algebra is adapted from Schmidt [Sch86], p 147. The only nonstandard operation is *makefresh*, which binds a list of locations in the store to fresh variables. This operation is used to construct the post-state store for executing a member function specification, as the locations that are bound to fresh variables are precisely the locations for which the post-condition defines new values according to the `modifies` clause in the member function specification.

Domain $Store = (Location \to Storable\text{-}value_{\perp}) \times Location$
Operations:
$newstore : \to Store$
$\quad newstore = (\lambda l.\perp, first\text{-}locn)$
$update : Location \to Storable\text{-}value \to Store \to Store$
$\quad update = \lambda l.\lambda v.\lambda(map, last).\ ([l \mapsto v]map, last)$
$access : Location \to Store \to Storable\text{-}value$
$\quad access = \lambda l.\lambda(map, last).\ map l$
$allocate\text{-}locn : Store \to Location \times Store$
$\quad allocate\text{-}locn = \lambda(map, last).\ (last, (map, next\text{-}locn(last)))$
$makefresh : Store \to List(Location) \to NextVar \to (Store \times NextVar)$
$\quad makefresh = \lambda(map, last).\lambda lst.\lambda nextv.\ \textbf{if}\ lst = nil\ \textbf{then}\ ((map, last), nextv)$
$\qquad \textbf{else let}\ (nv, nextv') = newvar(nextv)\ \textbf{in}$
$\qquad\quad makefresh([hd(l) \mapsto nv]map, last)\ tl(l)\ nextv'$

### 3.3 Constraints

Because constraints contain values (and not pieces of syntax), we cannot use elements of syntax domain Boolean-expr as constraints. Instead, 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

$\mathcal{C}$: Class-specification $\rightarrow$ *Environment* $\rightarrow$ *Class-Environment* $\rightarrow$ *PowerSet*(*Environment* $\times$ *Store*)

$\mathcal{C}$ $[\![$ `class` I `{`

```
/* model
  domains
    D
  data members
    A
  constraints
    B
  abstract functions
    F
*/
public:
  M
};
```

S $]\!] = \lambda e.\lambda ce.$

**let** $e' = \mathcal{F}[\![\mathrm{F}]\!]\ e$ **in**

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

$\mathcal{S}[\![\mathrm{S}]\!]\ (e', \mathit{updatecenv}\ \mathrm{I}\ (\mathcal{M}[\![\mathrm{M}]\!]\ e'\ \lambda i.\bot)\ ce, \{\}, \mathit{newstore}, \mathit{newnextv})$

Figure 4: Function $\mathcal{C}$ for evaluating a class specification.

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 that a set of constraints is inconsistent. We also define constraints representing universal quantification (*Forall*) and negation (*Not*). Figure 3 gives the definition of a representative subset of domain *Constraint*.

The domain *Constraint* will be used roughly as follows. First, some part of a SPECS-C++ specification is converted into a powerset of elements of *Constraint*, and then the elements of that powerset are simplified into sets of answers (environment and store pairs) using a fixpoint construction.

# 4 Valuation Functions

Figure 4 presents $\mathcal{C}$, the valuation function for class specifications.[1] The initial environment $e$ passed in can be the empty environment ($\lambda i.\bot$) if no "top level" definitions need to be provided. The initial class environment contains the member functions of any classes already processed. The denotations of the abstract functions are added to the environment by valuation function $\mathcal{F}$. The denotations of the member functions are stored in a new environment by $\mathcal{M}$ (see Figure 5 for both $\mathcal{F}$ and $\mathcal{M}$), which is then associated with the class name in the class environment. Both environments are used by $\mathcal{S}$ (Figure 6) to evaluate the "test case" (sequence of declarations, member function calls and assignment statements) for the specification. $\mathcal{S}$ returns a set of three-tuples (final environment, set of constraints, final store) as the set of it's final states. A set of states is necessary because specifications are often underdetermined or nondeterministic. If the set of constraints 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 $\mathcal{C}$. If the set of constraints 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. If the set of constraints contains in*False*(), then the algorithm determined that the constraints generated from the specification for this state are not satisfiable, and so the

[1] We use the notation $f^+$ throughout this paper to extend functions to deal with set-valued inputs. Given a function $f : T1 \rightarrow \mathit{PowerSet}(T2)$, $f^+ : \mathit{PowerSet}(T1) \rightarrow \mathit{PowerSet}(T2)$ is defined as: $f^+(X) = \bigcup\{f(x) \mid x \in X\}$.

$\mathcal{F}$: Abstract-function $\rightarrow$ *Environment* $\rightarrow$ *Environment*
$\mathcal{F}[\![\mathrm{F}_1; \mathrm{F}_2]\!] = \lambda e.\ \mathcal{F}[\![\mathrm{F}_2]\!]\ (\mathcal{F}[\![\mathrm{F}_1]\!]\ e)$
$\mathcal{F}[\![\texttt{define I(P) as T such that B}]\!] = \lambda e.$
$\quad$ **let** $f = \lambda l.\lambda c.\lambda s_{pre}.\lambda s_{post}.\lambda v.\lambda nextv.\ (\lambda(c', nextv').\ \{(c', s_{post}, nextv')\})^+$
$\qquad\qquad \mathcal{B}[\![\mathrm{B}]\!]\ (\mathcal{P}[\![P]\!]\ l\ e\ v, c, nextv)\ s_{pre}\ s_{post}$ **in**
$\qquad$ $updateenv\ [\![\mathtt{I}]\!]\ \mathrm{in}Function(f)\ e$
$\mathcal{F}[\![\epsilon]\!] = \lambda e.\ e$

$\mathcal{M}$: Member-function $\rightarrow$ *Environment* $\rightarrow$ *Class-Environment* $\rightarrow$ *Class-Environment*
$\mathcal{M}[\![\mathrm{M}_1; \mathrm{M}_2]\!] = \lambda e.\lambda ce.\ \mathcal{M}[\![\mathrm{M}_2]\!]\ e\ (\mathcal{M}[\![\mathrm{M}_1]\!]\ e\ ce)$
$\mathcal{M}$ [[ Y I(P);
$\quad$ `/* pre:` $\mathrm{B}_1$
$\quad\;$ `modifies:` L
$\quad\;$ `post:` $\mathrm{B}_2$`*/` $]\!] = \lambda e.\lambda ce.$
$\quad$ **let** $f = \lambda l.\lambda c.\lambda s_{pre}.\lambda s_{post}.\lambda v.\lambda nextv.$
$\qquad$ **let** $e' = updateenv\ [\![\mathtt{self}]\!]\ hd(l)\ (\mathcal{P}[\![\mathrm{P}]\!]\ tl(l)\ e\ v)$ **in**
$\qquad\quad (\lambda(e'', s', s'', c''', nextv'''').\ \{(c''', s'', nextv'''')\})^+$
$\qquad\qquad ((\mathit{fix}\ R^+)\ ((\lambda(lst, c', nextv').$ **let** $(s'_{post}, nextv'') = makefresh(s_{pre})\ lst\ nextv'$ **in**
$\qquad\qquad\quad (adjustState\ e'\ s_{pre}\ s'_{post}\ ((\lambda(c'', nextv''').\ \mathcal{B}[\![\mathrm{B}_2]\!]\ (e', c'', nextv''')\ s_{pre}\ s'_{post})^+$
$\qquad\qquad\qquad \mathcal{B}[\![\mathrm{B}_1]\!]\ (e', c', nextv'')\ s_{pre}\ s_{pre})))^+$
$\qquad\qquad\qquad\quad \mathcal{L}[\![\mathrm{L}]\!]\ (e', c, nextv)\ s_{pre}\ s_{pre}))$ **in**
$\qquad$ $updateenv\ [\![\mathtt{I}]\!]\ \mathrm{in}Function(f)\ ce$
$\mathcal{M}[\![\epsilon]\!] = \lambda e.\lambda ce.\ ce$

$\mathcal{P}$: Parameter $\rightarrow$ *List*(*Storable-value*) $\rightarrow$ *Environment* $\rightarrow$ *Storable-value* $\rightarrow$ *Environment*
$\mathcal{P}[\![\mathrm{P}_1; \mathrm{P}_2]\!] = \lambda l.\lambda e.\lambda v.\ \mathcal{P}[\![\mathrm{P}_2]\!]\ tl(l)\ (\mathcal{P}[\![\mathrm{P}_1]\!]\ cons(hd(l),\ nil)\ e\ v)\ v$
$\mathcal{P}[\![\mathrm{T\ I}]\!] = \lambda l.\lambda e.\lambda v.\ updateenv\ [\![I]\!]\ hd(l)\ e$
$\mathcal{P}[\![\epsilon]\!] = \lambda l.\lambda e.\lambda v.\ updateenv\ [\![\mathtt{result}]\!]\ v\ e$

Figure 5: Function $\mathcal{F}$ for storing abstract function definitions in an environment, function $\mathcal{M}$ for storing member function definitions, and function $\mathcal{P}$ that sets up the environment to evaluate the function body in.

environment and store should definitely not be part of the result as they will not satisfy the specification. For a specification using disjunction, set membership or existential quantification (constructs which may introduce additional states), it is likely that some states will contain in*False*() in the set of constraints. For example, the second set of constraints in the example given in the introduction would be simplified to a set containing in*False*(). If the set of constraints contains $\perp$, then the specification was partial or contained some error (see the explanation of Figure 8), and so does not contribute to the result. A set of constraints that is not empty and does not contain in*False*() or $\perp$ results from the incompleteness of the execution technique — the technique was not strong enough to simplify the remaining constraints.

Figure 5 presents $\mathcal{F}$ and $\mathcal{M}$, the valuation functions for abstract and member functions, respectively. In both cases, the function is compiled into a value of type *Function*, and then associated with its name in an environment so that it can be invoked later. Abstract functions are stored in the top level (global) environment, while the member functions of each class are stored in a class environment. Hence, $\mathcal{M}$ takes both the global environment and a class environment argument (that holds the member functions for the current class that have already been compiled), and returns a class environment. Function $\mathcal{P}$ is used to associate the formal parameters with the actuals in the environment used for evaluating the function's body. Note that parameter $v$ representing the function's result is associated with **`result`** in that environment. Abstract functions can call other abstract functions (and themselves recursively). We allow recursion by including the name of an abstract function in a *Call* constraint (see Figure 9) and then retrieving the function from the environment when simplifying such a constraint (see Figure 13). This delay ensures that

$\mathcal{S}$: Statement $\rightarrow$ ($\mathit{Environment} \times \mathit{Class\text{-}Environment} \times \mathit{PowerSet}(\mathit{Constraint}_\perp) \times \mathit{Store} \times \mathit{NextVar}$ ) $\rightarrow$
$\quad\mathit{PowerSet}((\mathit{Environment} \times \mathit{PowerSet}(\mathit{Constraint}_\perp) \times \mathit{Store} \times \mathit{NextVar}$ ))

$$
\begin{array}{l}
\mathcal{S}[\![\mathrm{S}_1;\mathrm{S}_2]\!] = \lambda(e,\ ce,\ c,\ s,\ nextv).\ (\lambda(e',\ c',\ s',\ nextv').\ \mathcal{S}[\![\mathrm{S}_2]\!]^+\ (e',\ ce,\ c',\ s',\ nextv'))^+ \\
\qquad\qquad\qquad (\mathcal{S}[\![\mathrm{S}_1]\!]\ (e,\ ce,\ c,\ s,\ nextv)) \\
\mathcal{S}[\![\mathrm{T\ I}]\!] = \lambda(e,\ ce,\ c,\ s,\ nextv). \\
\quad \mathbf{let}\ (s', l) = \mathit{allocate\text{-}locn}\ s\ \mathbf{in} \\
\quad\quad \mathbf{if}\ \mathrm{T} \in \mathrm{Identifier}\ \mathbf{then} \\
\quad\quad\quad \{(\mathit{updateenv}\ [\![\mathrm{I}]\!]\ \mathrm{in}\mathit{Value}(\mathrm{in}\mathit{Instance}(l,\mathrm{T}))\ e,\ c,\ s',\ nextv)\} \\
\quad\quad\quad \mathbf{else}\ \{(\mathit{updateenv}\ [\![\mathrm{I}]\!]\ \mathrm{in}\mathit{Value}(\mathrm{in}\mathit{Object}(l))\ e,\ c,\ s',\ nextv)\} \\
\mathcal{S}[\![\mathrm{I = E}]\!] = \lambda(e,\ ce,\ c,\ s,\ nextv). \\
\quad \mathbf{cases}\ \mathit{accessenv}\ [\![\mathrm{I}]\!]\ e\ \mathbf{of} \\
\quad\quad \mathrm{is}\mathit{Value}(v)\ \rightarrow \mathbf{cases}\ v\ \mathbf{of} \\
\quad\quad\quad \mathrm{is}\mathit{Object}(l)\ \rightarrow \\
\quad\quad\quad\quad \mathbf{let}\ (v', nextv') = \mathit{newvar}(nextv)\ \mathbf{in} \\
\quad\quad\quad\quad\quad (\lambda(e',c',s',s'',nextv'').\{(e',c',s'',nextv'')\})^+ \\
\quad\quad\quad\quad\quad\quad ((\mathit{fix}\ R^+)\ (\mathit{adjustState}\ e\ s\ (\mathit{update}\ l\ v'\ s)\ (\mathcal{E}[\![E]\!]\ v'\ (e,c,nextv')\ s\ s))) \\
\quad\quad\quad [\!]\ \mathrm{is}\mathit{Instance}(l,ci)\ \rightarrow \\
\quad\quad\quad\quad \mathbf{let}\ (v', nextv') = \mathit{newvar}(nextv)\ \mathbf{in} \\
\quad\quad\quad\quad\quad (\lambda(e',c',s',s'',nextv'').\{(e',c',s'',nextv'')\})^+ \\
\quad\quad\quad\quad\quad\quad ((\mathit{fix}\ R^+)\ (\mathit{adjustState}\ e\ s\ (\mathit{update}\ l\ v'\ s)\ (\mathcal{E}[\![E]\!]\ v'\ (e,c,nextv')\ s\ s))) \\
\quad\quad\quad \mathbf{else}\ \{(e,\ c \cup \{\perp\}, s, nextv)\} \\
\quad\quad \mathbf{end} \\
\quad\quad \mathbf{else}\ \{(e,\ c \cup \{\perp\}, s, nextv)\} \\
\quad \mathbf{end} \\
\mathcal{S}[\![\mathrm{I_1.I_2(L)}]\!] = \lambda(e,\ ce,\ c,\ s,\ nextv).\ \mathbf{cases}\ \mathit{accessenv}\ [\![\mathrm{I}_1]\!]\ e\ \mathbf{of} \\
\quad \mathrm{is}\mathit{Value}(v)\ \rightarrow \mathbf{cases}\ v\ \mathbf{of} \\
\quad\quad \mathrm{is}\mathit{Instance}(l,ci)\ \rightarrow \\
\quad\quad\quad \mathbf{cases}\ \mathit{accessenv}\ [\![\mathrm{I}_2]\!]\ (\mathit{accesscenv}\ ci\ ce)\ \mathbf{of} \\
\quad\quad\quad\quad \mathrm{is}\mathit{Function}(f)\ \rightarrow \\
\quad\quad\quad\quad\quad \mathbf{let}\ (nv, nextv') = \mathit{newvar}(nextv)\ \mathbf{in} \\
\quad\quad\quad\quad\quad\quad (\lambda(lst,\ c',\ nextv'').\ (\lambda(c'',\ s',\ nextv''').\ \{(e,\ c'',\ s',\ nextv''')\})^+ \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (f\ \ cons(v,\ lst)\ c'\ s\ s\ nv\ nextv''))^+ \\
\quad\quad\quad\quad\quad\quad\quad \mathcal{L}[\![L]\!]\ (e,\ c,\ nextv')\ s\ s \\
\quad\quad\quad\quad \mathbf{else}\ \{(e,\ c \cup \{\perp\}, s, nextv)\} \\
\quad\quad\quad \mathbf{end} \\
\quad\quad \mathbf{else}\ \{(e,\ c \cup \{\perp\}, s, nextv)\} \\
\quad \mathbf{end} \\
\quad \mathbf{else}\ \{(e,\ c \cup \{\perp\}, s, nextv)\} \\
\quad \mathbf{end} \\
\quad \vdots \\
\mathbf{end}
\end{array}
$$

Figure 6: Function $\mathcal{S}$ for evaluating statements that can call the member functions of the class specification. These statements include variable declarations and assignment statements. $\mathcal{S}$ uses function $\mathcal{L}$ (Figure 7) to evaluate actual parameter lists in member function calls.

$\mathcal{L}$: Expression-list $\rightarrow$ ( *Environment* $\times$ *PowerSet*( *Constraint*$_\perp$) $\times$ *NextVar* ) $\rightarrow$ *Store* $\rightarrow$ *Store*

$\quad\rightarrow$ *PowerSet*( *List*( *Storable-value*) $\times$ *PowerSet*( *Constraint*$_\perp$) $\times$ *NextVar* )

$$\begin{array}{l}
\mathcal{L}[\![\mathrm{L}_1;\mathrm{L}_2]\!] = \lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}. \\
\quad (\lambda(vl_1, c', nextv').\ (\lambda(vl_2, c'', nextv'').\ \{(append(lv_1, lv_2), c'', nextv'')\})^+ \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{L}[\![\mathrm{L}_2]\!]\ v\ (e, c', nextv')\ s_{pre}\ s_{post})^+ \\
\qquad \mathcal{L}[\![\mathrm{L}_1]\!]\ (e, c, nextv)\ s_{pre}\ s_{post} \\
\mathcal{L}[\![\mathrm{E}]\!] = \lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}. \\
\quad \textbf{let}\ (v, nextv') = newvar(nextv)\ \textbf{in} \\
\qquad (\lambda(c', nextv'').\ \{(cons(v, nil), c', nextv'')\})^+\ \mathcal{E}[\![E]\!]\ v\ (e, c, nextv')\ s_{pre}\ s_{post} \\
\mathcal{L}[\![\epsilon]\!] = \lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.\ \{(nil, c, nextv)\}
\end{array}$$

Figure 7: Function $\mathcal{L}$ evaluates a list of expressions into a list of variables and an associated constraint set.

all abstract functions are stored in the environment before any call to an abstract function is evaluated.

For member functions (valuation function $\mathcal{M}$), some additional work is required. The `modifies` clause specifies what objects can change from the pre-state to the post-state [GHG+93] [Win87]. This provides a "frame axiom" [BMR95] — if an object is not listed in the `modifies` clause, its value can not change, and so the specification need not define a value for that object. This list of objects is used by function *makefresh* (Section 3.2) to create the initial post-state store. Only objects listed in the `modifies` clause are bound to fresh variables, so attempts to change the values of other objects will cause the constraints generated from such a post-condition to be inconsistent. Note that the post-state store passed to the function is ignored in this case — this store is only passed in because it is needed for abstract functions. After constraints are generated from both the pre and post-condition, function $R$ (see Figure 10) 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. Function *adjustState* (see Figure 11) is used to put the arguments to $R^+$ into the proper form. Function $R$ 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. Member function specifications cannot be recursive — member functions can only be called from outside of the class.

Figure 6 presents representative cases of valuation function $\mathcal{S}$, which 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. The omitted case (for $\mathcal{S}[\![\mathrm{I}_1\ \texttt{=}\ \mathrm{I}_2.\mathrm{I}_3\texttt{(}\mathrm{L}\texttt{)}]\!]$) is just a combination of the last two cases shown. 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. Function $\mathcal{L}$ (Figure 7) is used to evaluate the actuals into a list of *Storable-value*s and an associated set of constraints to be passed to the denotation of the member function. Note that the object receiving the message (member function call) is packaged and passed as a regular parameter.

For simple assignment statements (assigning an expression result to an object), a new post-state store need not be allocated because such a statement can only change the value stored at one location. If the value being assigned to is not an object, then the return value has $\perp$ added to the constraint set to indicate the error. Many of the cases where $\perp$ is used represent type errors, and so the type checking done in the implementation would prevent these cases from occurring. We include them here for completeness, and in doing so, we allow $\perp$ to be an element of the sets of constraints returned from many of the functions in the semantics. We retain the constraints previously generated because we want conjunction to be nonstrict, and so a set of constraints that contains both in*False*() and $\perp$ is considered inconsistent and not an error case, as evidenced by Figure 12.

$\mathcal{B}$: Boolean-expr $\rightarrow$ ($Environment$ $\times$ $PowerSet(Constraint_\perp) \times NextVar$ ) $\rightarrow$ $Store \rightarrow Store \rightarrow$
$PowerSet(PowerSet(Constraint_\perp) \times NextVar)$

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

$\mathcal{B}[\![\texttt{false}]\!] = \lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.\ \{(\{\text{in}False()\}, nextv)\}$

$\mathcal{B}[\![\texttt{!}\mathrm{B}]\!] = \lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.\ \{(c\ \cup \{\text{in}Not(\mathrm{B}, e, s_{pre}, s_{post})\}, nextv)\}$

$\mathcal{B}[\![\mathrm{B}_1\ \texttt{/\textbackslash}\ \mathrm{B}_2]\!] = \lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.$
$\quad(\lambda(c', nextv').\ (\lambda(c'', nextv'').\ \{(c \cup c' \cup c'', nextv'')\})^+\ \mathcal{B}[\![\mathrm{B}_2]\!]\ (e, \{\}, nextv')\ s_{pre}\ s_{post})^+$
$\qquad\mathcal{B}[\![\mathrm{B}_1]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}$

$\mathcal{B}[\![\mathrm{B}_1\ \texttt{\textbackslash/}\ \mathrm{B}_2]\!] = \lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.\ \mathcal{B}[\![\mathrm{B}_1]\!]\ (e, c, nextv)\ s_{pre}\ s_{post}\ \cup \mathcal{B}[\![\mathrm{B}_2]\!]\ (e, c, nextv)\ s_{pre}\ s_{post}$

$\mathcal{B}[\![\mathrm{B}_1\ \texttt{=>}\ \mathrm{B}_2]\!] = \mathcal{B}[\![(\mathrm{B}_1\ \texttt{/\textbackslash}\ \mathrm{B}_2)\ \texttt{\textbackslash/}\ \texttt{!}\mathrm{B}_1]\!]$

$\mathcal{B}[\![\texttt{\textbackslash forall}\ \mathrm{T\ I}\ \texttt{[(}\mathrm{I}\ \texttt{\textbackslash in}\ \mathrm{E}\ \texttt{/\textbackslash}\ \mathrm{B}_1\texttt{)}\ \texttt{=>}\ \mathrm{B}_2\ \texttt{]}]\!] = \lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.$
$\quad\textbf{let}\ (v, nextv') = newvar(nextv)\ \textbf{in}$
$\qquad(\lambda(c', nextv'').\ \{(c'\ \cup\ \{\text{in}Forall(\mathrm{I}, v, \mathrm{B}_1\ \texttt{=>}\ \mathrm{B}_2, e, s_{pre}, s_{post})\}, nextv'')\})^+$
$\qquad\quad\mathcal{E}[\![\mathrm{E}]\!]\ v\ (e, c, nextv')\ s_{pre}\ s_{post}$

$\mathcal{B}[\![\texttt{\textbackslash exists}\ \mathrm{T\ I}\ \texttt{[B]}]\!] = \lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.$
$\quad\textbf{let}\ (v, nextv') = newvar(nextv)\ \textbf{in}$
$\qquad\mathcal{B}[\![\mathrm{B}]\!]\ (updateenv\ [\![I]\!]\ v\ e, c, nextv')\ s_{pre}\ s_{post}$

$\mathcal{B}[\![\mathrm{E}_1\ \texttt{=}\ \mathrm{E}_2]\!] = \lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.$
$\quad\textbf{let}\ (v_1, nextv') = newvar(nextv)\ \textbf{in}$
$\quad\textbf{let}\ (v_2, nextv'') = newvar(nextv')\ \textbf{in}$
$\qquad(\lambda(c', nextv''').\ (\lambda(c'', nextv'''').\ \{(c'' \cup \{\text{in}Equal(v_1, v_2)\}, nextv'''')\})^+$
$\qquad\qquad\mathcal{E}[\![\mathrm{E}_2]\!]\ v_2\ (e, c', nextv''')\ s_{pre}\ s_{post})^+$
$\qquad\quad\mathcal{E}[\![\mathrm{E}_1]\!]\ v_1\ (e, c, nextv'')\ s_{pre}\ s_{post}$

$\quad\vdots$

**end**

Figure 8: Representative cases of valuation function $\mathcal{B}$ for evaluating boolean assertions into sets of constraints.

The fixpoint construction used earlier must be applied here as well to ensure that the set of constraints is simplified if the assignment statement is the last statement in the test case. Assignment statements are the only case requiring constraint simplification where a member function is not invoked, and so we cannot rely on the execution of a member function to apply the fixpoint construction in this case.

Figure 8 presents valuation function $\mathcal{B}$, which is used to translate assertions into constraints. 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. A *Not* constraint will be created as part of the evaluation of some member function call from some statement in the test cases. The evaluation of the constraint might be delayed until some following statement is being evaluated, in which case the current post-state store would not be the same as the post-state store in effect at the time that the *Not* constraint was created.

For translating conjunctions, first one conjunct is translated, and the resulting constraint set is used as input to the translation of the next conjunct. This ensures that the constraints generated from both conjuncts are in the same set of constraints. The semantics of a set of constraints is conjunction — all constraints in the set must consistent for the set to be consistent. By *consistent*, we mean that there exists some assignment of values to the variables in the constraints that makes all of them true simultaneously. For disjunctions, each disjunct is translated and the resulting sets of constraints and *NextVar* values are added to the powerset of constraints and *NextVar* values. The constraints from each disjunct are kept in separate sets, so that if one set is not consistent, the disjunction can still be satisfiable (if the constraints generated from the other disjunct are consistent). The semantics of a powerset of constraints is disjunction — the powerset is consistent if any set it contains is consistent. The advantage of this approach is that it allows us to handle underdetermined or nondeterministic specifications explicitly — nondeterminism simply produces multiple sets of constraints and *NextVar* values. As we will prove in Section 5, an assertion is satisfiable when any of the sets of constraints and *NextVar* values generated from it are consistent.

The semantics uses a three-valued logic internally. This is not apparent externally, as valuation function $\mathcal{C}$ (Figure 4), which provides the external interface to the semantics, produces no output for constraint sets that contain $\perp$. However, some of the built-in operations of SPECS-C++ are partial (not defined for all inputs), and specifications can be partial in general, and so the satisfiability of an assertion may be undefined [BCJ84] [JM94]. Both conjunction and disjunction are nonstrict, and the order that the specification is processed in does not matter (except for termination). That is, if any disjunct is satisfiable, the powerset of constraints generated from the entire disjunction is consistent, even though some other disjuncts may be undefined (evaluate to a constraint set containing $\perp$). Similarly, if any conjunct is not satisfiable, the entire conjunction is not satisfiable, even if some other conjunct is undefined. We enforce this behavior by ensuring that any constraint set that contains in*False*() is considered inconsistent (see the is*Not* case in Figure 12, for example), even if that constraint set also contains $\perp$. Our implementation also behaves in this way, which is an advantage over some other executable specification languages such as IFAD VDM-SL [ELL94], which executes expressions from left to right, so that "true or undefined" valuates to "true", while "undefined or true" evaluates to "undefined".

An implication `P => Q` is translated as the equivalent `(P /\ Q) \/ !P`. The reason that this translation is favored over the standard `!P \/ Q` is that a *Not* constraint can never be used to directly define the post-state store (again, see the is*Not* case in Figure 12), so the standard translation would never allow constraints generated from `P` to be used in this way.

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. Here, the domain is evaluated, and included in a *Forall* constraint to be evaluated later. Existential quantification simply introduces a new variable.

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 $\mathcal{E}$ (Figure 9) 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: $\{\text{in}Times(x, 2, v), \text{in}Plus(v, y, v')\}$, 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 possible — no propagation can be 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 $\mathcal{E}$ are similar, and in fact most are omitted from Figure 9. 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 (term) 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 `!`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 `!`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.

The function $R$ (Figure 10) is used to simplify a set of constraints into a store. We refer to an element of type $Environment \times Store \times Store \times PowerSet(Constraint_\perp) \times NextVar$ as a computation state, since it captures the progress made so far in simplifying constraints. The second *Store* component represents the post-state store, so each successful application of $R$ simplifies the set of constraints component by removing some constraints and/or replacing some constraints with simpler ones, and can cause the second *Store* component to become more defined. If an application of $R$ fails (for example, if the set of constraints

$\mathcal{E}$: Expression $\rightarrow$ *Storable-value* $\rightarrow$ $(\mathit{Environment} \times \mathit{PowerSet}(\mathit{Constraint}_\perp) \times \mathit{NextVar}) \rightarrow \mathit{Store} \rightarrow \mathit{Store}$
$\rightarrow \mathit{PowerSet}(\mathit{PowerSet}(\mathit{Constraint}_\perp) \times \mathit{NextVar})$

$\mathcal{E}[\![\mathrm{B}]\!] = \lambda v.\lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.$
**cases** $v$ **of**
is$\mathit{Value}(b)$ $\rightarrow$
**if** $b$ **then** $\mathcal{B}[\![\mathrm{B}]\!]\ (e, c, nextv)\ s_{pre}\ s_{post}$ **else** $\mathcal{B}[\![\texttt{!}\mathrm{B}]\!]\ (e, c, nextv)\ s_{pre}\ s_{post}$
$[\!]$ is$\mathit{Variable}(i)$ $\rightarrow$
$\mathcal{B}[\![\mathrm{B}]\!]\ (e, c \cup \{\mathrm{in}\mathit{Equal}(v, \mathrm{in}\mathit{Value}(\mathrm{in}\mathit{Boolean}(\mathit{true})))\}, nextv)\ s_{pre}\ s_{post}$
$\cup\ \mathcal{B}[\![\texttt{!}\mathrm{B}]\!]\ (e, c \cup \{\mathrm{in}\mathit{Equal}(v, \mathrm{in}\mathit{Value}(\mathrm{in}\mathit{Boolean}(\mathit{false})))\}), nextv)\ s_{pre}\ s_{post}$
**end**

$\mathcal{E}[\![\mathrm{I}]\!] = \lambda v.\lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.$
**let** $v' = \mathit{accessenv}\ [\![\mathrm{I}]\!]\ e$ **in**
**if** $v' = \perp$ **then** $\{(c \cup \{\perp\}, nextv)\}$
**else** $\{(c \cup \{\mathrm{in}\mathit{Equal}(v, v')\}, nextv)\}$

$\mathcal{E}[\![\texttt{first(E)}]\!] = \lambda v.\lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.$
**let** $(v', nextv') = \mathit{newvar}(nextv)$ **in**
$(\lambda(c', nextv'').\ \{(c' \cup \{\mathrm{in}\mathit{First}(v', v)\}, nextv'')\})^+\ \mathcal{E}[\![\mathrm{E}]\!]\ v'\ (e, c, nextv')\ s_{pre}\ s_{post}$

$\mathcal{E}[\![\mathrm{E}_1\ \texttt{+}\ \mathrm{E}_2]\!] = \lambda v.\lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.$
**let** $(v_1, nextv') = \mathit{newvar}(nextv)$ **in**
**let** $(v_2, nextv'') = \mathit{newvar}(nextv')$ **in**
$(\lambda(c', nextv''').\ (\lambda(c'', nextv'''').\ \{(c'' \cup \{\mathrm{in}\mathit{Plus}(v_1, v_2, v)\}, nextv'''')\})^+$
$\mathcal{E}[\![\mathrm{E}_2]\!]\ v_2\ (e, c', nextv''')\ s_{pre}\ s_{post})^+$
$\mathcal{E}[\![\mathrm{E}_1]\!]\ v_1\ (e, c, nextv'')\ s_{pre}\ s_{post}$

$\mathcal{E}[\![\mathrm{E}\texttt{'}]\!] = \lambda v.\lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.$
**let** $(v', nextv') = \mathit{newvar}(nextv)$ **in**
$(\lambda(c', nextv'').\ \{(c' \cup \{\mathrm{in}\mathit{Post}(v', v)\}, nextv'')\})^+\ \mathcal{E}[\![\mathrm{E}]\!]\ v'\ (e, c, nextv')\ s_{pre}\ s_{post}$

$\mathcal{E}[\![\texttt{result}]\!] = \lambda v.\lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.\ \{(c \cup \{\mathrm{in}\mathit{Equal}(v, \mathit{accessenv}\ [\![\texttt{result}]\!]\ e)\}, nextv)\}$

$\mathcal{E}[\![\texttt{self}]\!] = \lambda v.\lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.\ \{(c \cup \{\mathrm{in}\mathit{Equal}(v, \mathit{accessenv}\ [\![\texttt{self}]\!]\ e)\}, nextv)\}$

$\mathcal{E}[\![\texttt{\{T I : I \textbackslash in E /\textbackslash\ B\}}]\!] = \lambda v.\lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.$
**let** $(v', nextv') = \mathit{newvar}(nextv)$ **in**
$(\lambda(c', nextv'').\ \{(c' \cup \{\mathrm{in}\mathit{Comp}(\mathrm{I}, v', \mathrm{B}, v, e, s_{pre}, s_{post})\}, nextv'')\})^+$
$\mathcal{E}[\![\mathrm{E}]\!]\ v'\ (e, c, nextv')\ s_{pre}\ s_{post}$

$\mathcal{E}[\![\texttt{I(L)}]\!] = \lambda v.\lambda(e, c, nextv).\lambda s_{pre}.\lambda s_{post}.$
$(\lambda(lv, c', nextv').\ \{(c' \cup \{\mathrm{in}\mathit{Call}(\mathrm{I}, lv, v, s_{pre}, s_{post})\}, nextv')\}\})^+$
$\mathcal{L}[\![\mathrm{L}]\!]\ (e, c, nextv)\ s_{pre}\ s_{post}$
⋮
**end**

Figure 9: Representative cases of valuation function $\mathcal{E}$.

$R : (\mathit{Environment} \times \mathit{Store} \times \mathit{Store} \times \mathit{PowerSet}(\mathit{Constraint}_\perp) \times \mathit{NextVar})$
$\rightarrow \mathit{PowerSet}(\mathit{Environment} \times \mathit{Store} \times \mathit{Store} \times \mathit{PowerSet}(\mathit{Constraint}_\perp) \times \mathit{NextVar})$

$R = \lambda(e, s_{pre}, s_{post}, cs, nextv).$ **if** $cs = \{\}$ **then** $\{(e, s_{pre}, s_{post}, cs, nextv)\}$
**else** $\bigcup\{S\ c\ (e, s_{pre}, s_{post}, cs - \{c\}, nextv) \mid c \in cs\}$

Figure 10: The function $R$ used for simplifying constraints. The least fixed point of $R^+$ is the denotation of an assertion. $R$ uses function $S$ (defined in Figures 12 and 13) to simplify each constraint.

contains inconsistent constraints), the set of constraints will be simplified to a set containing in*False*() to indicate this failure.

$R$ 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 12 and 13. For each kind of constraint, $S$ determines if enough information is available to make a simplification step. If not, then $S$ leaves the set of constraints unchanged (by returning the constraint to the set). Otherwise, $S$ performs a simplification, in which case the constraint used is not returned to the set of constraints. 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 local 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.

For *Forall* constraints, $S$ first checks if the domain of the bound variable (represented by $ds$) is known. If so, then if the domain is the empty set, the constraint is equivalent to *true*, and so is simply not included in the return constraint set. Otherwise, some element $v$ of the domain is chosen, and constraints are generated from the body of the assertion (B) in an environment with the bound variable I mapped to $v$. Additionally, three constraints are placed with the constraints just generated in the return constraint set. These constraints have the effect of removing $v$ from the domain of the bound variable in the *Forall* constraint.

For *Not* constraints, $S$ generates and simplifies all states (with contained constraint sets) from the body of the negated assertion B. If some constraint set so produced is empty, the body of the negated assertion is true, and so the assertion is false. The resulting constraint set contains a *False* constraint, and so is inconsistent. If all constraint sets so produced are inconsistent, then the body of the negated assertion is false, and so the assertion is true. Hence, the *Not* constraint is simply not included in the return constraint set. Otherwise, the truth value of the body of the negated assertion can not be determined, and so the constraint set returned is the same as the input constraint set.

For *Post* constraints, if the first component of the constraint is a location (object or instance), then the value stored at that location in the post-state store is equated with the second component of the constraint. For *Call* constraints, the denotation of the matching abstract function is retrieved from the environment and invoked, producing a set of states.

To ensure that the least fixed point of $R^+$ exists, we need to establish that the domain of $R^+$ is a pointed complete partial ordering (cpo), and that the function $R^+$ is continuous. The domain of $R^+$ is $D = \mathit{PowerSet}(\mathit{Environment} \times \mathit{Store} \times \mathit{Store} \times \mathit{PowerSet}(\mathit{Constraint}) \times \mathit{NextVar})$ — i.e., sets of sets of computation states. We define the partial order relation $\sqsubseteq$ on this domain $D$ as $\subseteq$, and so the least element is $\{\}$. One complications is that we can not use ordinary equality when testing that one set in $D$ is a subset of another. Two applications of $R$ to the same computation state can result in syntactically different results because of the order that the constraints in the set of constraints are processed in. This occurs because of or use of type *NextVar* in generating fresh variables. Since we essentially have a global counter for generating variable names, the variable names contained in a constraint depend on the order that constraints are processed in. Hence, we define two computation states to be equal if they are equal up to renaming of variables, i.e. if there is a renaming of the variables of one computation state that converts it to the other. In this case, the renaming in the reverse direction must also exist. This definition is consistent with the meaning of a computation state, as variable names do not matter. We also ignore the value of the *NextVar* parameter. Our use of renaming in defining equality for the rest of the computation state means that the exact variable names generated using this component are insignificant. Hence, we are implicitly defining $\sqsubseteq$ for equivalence classes of domain elements, rather than for domain elements directly. If we use this definition of equality for evaluating operations $\cup$ and $\cap$, we can then define join ($\sqcup$) as $\cup$ and meet ($\sqcap$) as $\cap$. Then $D$ is a complete lattice, as for any subset $X$ of $D$, both $\bigsqcup X$ and $\bigsqcap X$ exist. Since $D$ is a complete lattice and has a least element $\{\}$, $D$ is a pointed cpo.

$adjustState : Environment \rightarrow Store \rightarrow Store \rightarrow PowerSet(PowerSet(Constraint_\perp) \times NextVar)$
$\quad \rightarrow PowerSet(Environment \times Store \times Store \times PowerSet(Constraint_\perp) \times NextVar)$
$adjustState = \lambda e.\lambda s_{pre}.\lambda s_{post}.\lambda cs.\ \{(e, s_{pre}, s_{post}, c, nextv) \mid (c, nextv) \in cs\}$

Figure 11: Function *adjustState*, which is used to adjust the representation of a state to the form needed for applying $R$.

Next, we need to establish that $R^+$ is continuous. To show that $R^+$ is monotonic, we simply note that for $x, y \in D$, if $x \sqsubseteq y$, then $R^+(x) \sqsubseteq R^+(y)$ must hold, as for each element of $R^+(x)$, there will be an equivalent value in $R^+(y)$, since there is an equivalent element in $y$ for each element in $x$. To show that $R^+$ is continuous, we need to show that for any chain $X$ in $D$, $R^+(\bigsqcup X)$ is equivalent to $\bigsqcup\{R^+(x) \mid x \in X\}$. To show that $\bigsqcup\{R^+(x) \mid x \in X\} \sqsubseteq R^+(\bigsqcup X)$:

$$\begin{array}{ll} & true \\ \Rightarrow & \langle \text{by the definition of } \bigsqcup \rangle \\ & \forall x \in X, x \sqsubseteq \bigsqcup X \\ \Rightarrow & \langle \text{by the monotonicity of } R^+ \rangle \\ & \forall x \in X, R^+(x) \sqsubseteq R^+(\bigsqcup X) \\ \Rightarrow & \langle \text{by the definition of } \bigsqcup \rangle \\ & \bigsqcup\{R^+(x) \mid x \in X\} \sqsubseteq R^+(\bigsqcup X) \end{array}$$

To show that $R^+(\bigsqcup X) \sqsubseteq \bigsqcup\{R^+(x) \mid x \in X\}$, let $y$ be an arbitrary state such that $y \in R^+(\bigsqcup X)$. We show that $y \in \bigsqcup\{R^+(x) \mid x \in X\}$. Since $y \in R^+(\bigsqcup X)$, by the definition of $R^+$ there must exist some state $z \in \bigsqcup X$ such that $y \in R^+(\{z\})$. By the definition of $\bigsqcup$ (as *least* upper bound), there must exist some $x' \in X$ such that $z \in x'$. Since $z \in x'$, $y \in R^+(x')$ by the definition of $R^+$. Since $x' \in X$, $y \in \bigsqcup\{R^+(x) \mid x \in X\}$. Hence, $R^+(\bigsqcup X) \sqsubseteq \bigsqcup\{R^+(x) \mid x \in X\}$.

Since $R^+$ is continuous and its domain $D$ is a pointed cpo, the least fixed point of $R^+$ exists.

# 5 Proof of Soundness

In this section, we concentrate on proving the soundness of valuation functions $\mathcal{B}$ (Figure 8) and $\mathcal{E}$ (Figure 9), and on proving the soundness of $(\mathit{fix}\ R^+)$ (Figure 10). These are the parts of the semantics that deal directly with assertions, and so we can show soundness by showing that any environment and store produced by the semantics satisfies the assertion that was input.

The valuation functions that precede $\mathcal{B}$ ($\mathcal{C}$, $\mathcal{M}$, $\mathcal{F}$ and $\mathcal{S}$ in Figures 4 through 6) 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 here.

We state the soundness result for assertions as a theorem below. The notation B$[e, s_{pre}, s'_{post}]$ means the truth value of B under the given environment and stores.

**Theorem 5.1** *Soundness Theorem:*

$$\begin{array}{l} \forall B \in \mathit{Boolean\text{-}expr}.\ \forall (cs, nextv') \in \mathcal{B}[\![B]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}. \\ \quad \forall (e', s'_{pre}, s'_{post}, cs', nextv'') \in (\mathit{fix}\ R^+)\ \{(e, s_{pre}, s_{post}, cs, nextv')\}. \\ \qquad cs' = \{\} \Rightarrow B[e, s_{pre}, s'_{post}] \end{array}$$

The theorem states that if the constraints derived from B are simplified into the environment $e'$, pair of stores $s'_{pre}$ and $s'_{post}$, and set of constraints $cs'$, and all constraints can be simplified ($cs'$ is empty), then the assertion B is true under $e$, $s_{pre}$ and $s'_{post}$. We ignore $s'_{pre}$ 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.

$$
\begin{array}{l}
S : \mathit{Constraint}_\perp \to (\mathit{Environment} \times \mathit{Store} \times \mathit{Store} \times \mathit{PowerSet}(\mathit{Constraint}_\perp) \times \mathit{NextVar}) \\
\quad \to \mathit{PowerSet}(\mathit{Environment} \times \mathit{Store} \times \mathit{Store} \times \mathit{PowerSet}(\mathit{Constraint}_\perp) \times \mathit{NextVar}) \\
S = \lambda c.\lambda(e, s_{pre}, s_{post}, cs, nextv). \\
\quad \textbf{if } c = \perp \textbf{ then } \{(e, s_{pre}, s_{post}, cs \cup \{\perp\}, nextv)\} \\
\quad \textbf{else cases } c \textbf{ of} \\
\qquad \text{is}\mathit{Forall}(\mathrm{I}, ds, \mathrm{B}, e', s'_{pre}, s'_{post}) \to \textbf{cases } ds \textbf{ of} \\
\qquad\qquad \text{is}\mathit{Value}(d) \to \textbf{cases } d \textbf{ of} \\
\qquad\qquad\quad \text{is}\mathit{Set}(vl) \to (\textbf{if } vl = nil \textbf{ then } \{(e, s_{pre}, s_{post}, cs, nextv)\} \\
\qquad\qquad\qquad\qquad \textbf{else let } (v, nextv') = \mathit{newvar}(nextv) \textbf{ in} \\
\qquad\qquad\qquad\qquad \textbf{let } e'' = \mathit{updateenv}\ [\![\mathrm{I}]\!]\ v\ e' \textbf{ in} \\
\qquad\qquad\qquad\qquad \textbf{let } (v', nextv'') = \mathit{newvar}(nextv') \textbf{ in} \\
\qquad\qquad\qquad\qquad\quad (\lambda(cs', nextv'''). \{(e, s_{pre}, s_{post}, \\
\qquad\qquad\qquad\qquad\qquad cs' \cup \{\text{in}\mathit{Member}(v, ds), \text{in}\mathit{Minus}(ds, v, v'), \\
\qquad\qquad\qquad\qquad\qquad\ \ \text{in}\mathit{Forall}(\mathrm{I}, v', \mathrm{B}, e', s'_{pre}, s'_{post})\}, \\
\qquad\qquad\qquad\qquad\qquad nextv''')\})^{+}\ \mathcal{B}[\![\mathrm{B}]\!]\ (e'', cs, nextv'')\ s'_{pre}\ s'_{post} \\
\qquad\qquad\quad \textbf{else } \{(e, s_{pre}, s_{post}, cs \cup \{\perp\}, nextv)\} \\
\qquad\qquad \textbf{end} \\
\qquad\quad [\!]\ \text{is}\mathit{Variable}(i) \to \{(e, s_{pre}, s_{post}, cs \cup \{\text{in}\mathit{Forall}(\mathrm{I}, ds, \mathrm{B}, e', s'_{pre}, s'_{post})\}, nextv)\} \\
\qquad \textbf{end} \\
\quad [\!]\ \text{is}\mathit{Not}(\mathrm{B}, e', s'_{pre}, s'_{post}) \to \\
\qquad\quad \textbf{let } res = (\mathit{fix}\ R^{+})\ (\mathit{adjustState}\ e\ s_{pre}\ s_{post}\ (\mathcal{B}[\![\mathrm{B}]\!]\ (e', cs, nextv)\ s'_{pre}\ s'_{post})) \textbf{ in} \\
\qquad\qquad \textbf{if } \exists(e', s''_{pre}, s''_{post}, cs', nextv') \in res\ .\ cs' = \{\} \textbf{ then} \\
\qquad\qquad\quad \{(e, s_{pre}, s_{post}, \{\text{in}\mathit{False}()\}, nextv')\} \\
\qquad\qquad \textbf{else if } \forall(e', s''_{pre}, s''_{post}, cs', nextv') \in res\ .\ \text{in}\mathit{False}() \in cs' \textbf{ then} \\
\qquad\qquad\quad \{(e, s_{pre}, s_{post}, cs, nextv')\} \\
\qquad\qquad \textbf{else } \{(e, s_{pre}, s_{post}, cs \cup \{\text{in}\mathit{Not}(\mathrm{B}, e', s'_{pre}, s'_{post})\}, nextv)\} \\
\quad [\!]\ \text{is}\mathit{False}() \to \{(e, s_{pre}, s_{post}, \{\text{in}\mathit{False}()\}, nextv)\} \\
\quad [\!]\ \text{is}\mathit{Equal}(v_1, v_2) \to \textbf{cases } v_1 \textbf{ of} \\
\qquad\quad \text{is}\mathit{Variable}(i) \to \{([v_2/v_1]\ e, s_{pre}, [v_2/v_1]\ s_{post}, [v_2/v_1]\ cs, nextv)\} \\
\qquad [\!]\ \text{is}\mathit{Value}(n) \to \textbf{cases } v_2 \textbf{ of} \\
\qquad\qquad \text{is}\mathit{Variable}(i') \to \{([v_1/v_2]\ e, s_{pre}, [v_1/v_2]\ s_{post}, [v_1/v_2]\ cs, nextv)\} \\
\qquad\quad [\!]\ \text{is}\mathit{Value}(n') \to \textbf{let } (cs', nextv') = \mathit{unify}(v_1, v_2, cs, nextv) \textbf{ in } \{(e, s_{pre}, s_{post}, cs', nextv')\} \\
\qquad\quad \textbf{end} \\
\qquad \textbf{end} \\
\quad [\!]\ \text{is}\mathit{First}(v_1, v_2) \to \textbf{cases } v_1 \textbf{ of} \\
\qquad\quad \text{is}\mathit{Variable}(i) \to \textbf{cases } v_2 \textbf{ of} \\
\qquad\qquad \text{is}\mathit{Variable}(i') \to \{(e, s_{pre}, s_{post}, cs \cup \{\text{in}\mathit{First}(v_1, v_2)\}, nextv)\} \\
\qquad\quad [\!]\ \text{is}\mathit{Value}(el) \to \textbf{let } (nv, nextv') = \mathit{newvar}(nextv) \textbf{ in} \\
\qquad\qquad\quad \{(e, s_{pre}, s_{post}, cs \cup \{\text{in}\mathit{Concat}(\text{in}\mathit{Value}(\text{in}\mathit{Sequence}(\mathit{cons}(v_2, nil))), nv, v_1)\}, nextv')\} \\
\qquad\quad \textbf{end} \\
\qquad [\!]\ \text{is}\mathit{Value}(seq) \to \textbf{cases } seq \textbf{ of} \\
\qquad\qquad \text{is}\mathit{Sequence}(l) \to \{(e, s_{pre}, s_{post}, cs \cup \{\text{in}\mathit{Equal}(v2, hd(l))\}, nextv)\} \\
\qquad\qquad \textbf{else } \{(e, s_{pre}, s_{post}, cs \cup \{\perp\}, nextv)\} \\
\qquad\quad \textbf{end} \\
\qquad \textbf{end}
\end{array}
$$

Figure 12: Representative cases of function $S$, which defines the effect of simplifying a single constraint. (Part 1 of 2.) Function *adjustState* is defined in Figure 11.

$$
\begin{array}{l}
\;[\!]\ \mathrm{is}\mathit{Plus}(v_1, v_2, v_3)\ \rightarrow \\
\quad\quad \mathbf{cases}\ v_1\ \mathbf{of} \\
\quad\quad\quad\quad \mathrm{is}\mathit{Variable}(i_1)\ \rightarrow\ \mathbf{cases}\ v_2\ \mathbf{of} \\
\quad\quad\quad\quad\quad\quad \mathrm{is}\mathit{Variable}(i_2)\ \rightarrow\ \{(e, s_{pre}, s_{post}, cs \cup \{\mathrm{in}\mathit{Plus}(v_1, v_2, v_3)\}, nextv)\} \\
\quad\quad\quad\quad\quad [\!]\ \mathrm{is}\mathit{Value}(n_2)\ \rightarrow\ \mathbf{cases}\ v_3\ \mathbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathrm{is}\mathit{Variable}(i_3)\ \rightarrow\ \{(e, s_{pre}, s_{post}, cs \cup \{\mathrm{in}\mathit{Plus}(v_1, v_2, v_3)\}, nextv)\} \\
\quad\quad\quad\quad\quad\quad\quad [\!]\ \mathrm{is}\mathit{Value}(n_3)\ \rightarrow\ \{(e, s_{pre}, s_{post}, cs \cup \{\mathrm{in}\mathit{Equal}(v_1, \mathrm{in}\mathit{Value}(n_3 - n_2)))\}, nextv)\} \\
\quad\quad\quad\quad\quad\quad \mathbf{end} \\
\quad\quad\quad\quad \mathbf{end} \\
\quad\quad\quad [\!]\ \mathrm{is}\mathit{Value}(n_1)\ \rightarrow\ \mathbf{cases}\ v_2\ \mathbf{of} \\
\quad\quad\quad\quad\quad\quad \mathrm{is}\mathit{Variable}(i_2)\ \rightarrow\ \mathbf{cases}\ v_3\ \mathbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathrm{is}\mathit{Variable}(i_3)\ \rightarrow\ \{(e, s_{pre}, s_{post}, cs \cup \{\mathrm{in}\mathit{Plus}(v_1, v_2, v_3)\}, nextv)\} \\
\quad\quad\quad\quad\quad\quad\quad [\!]\ \mathrm{is}\mathit{Value}(n_3)\ \rightarrow\ \{(e, s_{pre}, s_{post}, cs \cup \{\mathrm{in}\mathit{Equal}(v_2, \mathrm{in}\mathit{Value}(n_3 - n_1)))\}, nextv)\} \\
\quad\quad\quad\quad\quad\quad \mathbf{end} \\
\quad\quad\quad\quad\quad [\!]\ \mathrm{is}\mathit{Value}(n_2)\ \rightarrow\ \{(e, s_{pre}, s_{post}, cs \cup \{\mathrm{in}\mathit{Equal}(v_3, \mathrm{in}\mathit{Value}(n_1 + n_2)))\}, nextv)\} \\
\quad\quad\quad\quad \mathbf{end} \\
\quad\quad \mathbf{end} \\
\;[\!]\ \mathrm{is}\mathit{Post}(v_1, v_2)\ \rightarrow\ \mathbf{cases}\ v_1\ \mathbf{of} \\
\quad\quad\quad\quad \mathrm{is}\mathit{Variable}(i_1)\ \rightarrow\ \{(e, s_{pre}, s_{post}, cs \cup \{\mathrm{in}\mathit{Post}(v_1, v_2)\}, nextv)\} \\
\quad\quad\quad [\!]\ \mathrm{is}\mathit{Value}(l)\ \rightarrow\ \mathbf{cases}\ l\ \mathbf{of} \\
\quad\quad\quad\quad\quad\quad \mathrm{is}\mathit{Object}(l')\ \rightarrow\ \mathbf{let}\ v' = \mathit{access}\ l'\ s_{post}\ \mathbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{if}\ v' = \bot\ \mathbf{then}\ \{(e, s_{pre}, s_{post}, cs \cup \{\bot\}, nextv)\} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{else}\ \{(e, s_{pre}, s_{post}, cs \cup \{\mathrm{in}\mathit{Equal}(v', v_2)\}, nextv)\} \\
\quad\quad\quad\quad\quad [\!]\ \mathrm{is}\mathit{Instance}(l', ci)\ \rightarrow\ \mathbf{let}\ v' = \mathit{access}\ l'\ s_{post}\ \mathbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{if}\ v' = \bot\ \mathbf{then}\ \{(e, s_{pre}, s_{post}, cs \cup \{\bot\}, nextv)\} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{else}\ \{(e, s_{pre}, s_{post}, cs \cup \{\mathrm{in}\mathit{Equal}(v', v_2)\}, nextv)\} \\
\quad\quad\quad\quad\quad\quad \mathbf{else}\ \{(e, s_{pre}, s_{post}, cs \cup \{\bot\}, nextv)\} \\
\quad\quad\quad\quad \mathbf{end} \\
\quad\quad \mathbf{end} \\
\;[\!]\ \mathrm{is}\mathit{Comp}(\mathrm{I}, v_1, \mathrm{B}, v_2, e', s'_{pre}, s'_{post})\ \rightarrow \\
\quad\quad \mathbf{cases}\ v_1\ \mathbf{of} \\
\quad\quad\quad\quad \mathrm{is}\mathit{Variable}(i)\ \rightarrow\ \{(e, s_{pre}, s_{post}, cs \cup \{\mathrm{in}\mathit{Comp}(\mathrm{I}, v_1, \mathrm{B}, v_2, e', s'_{pre}, s'_{post})\}, nextv)\} \\
\quad\quad\quad [\!]\ \mathrm{is}\mathit{Value}(s)\ \rightarrow\ \mathbf{cases}\ s\ \mathbf{of} \\
\quad\quad\quad\quad\quad\quad \mathrm{is}\mathit{Set}(vl)\ \rightarrow\ (\lambda(vl', cs', nextv').\ \{(e, s_{pre}, s_{post}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad cs' \cup \{\mathrm{in}\mathit{Equal}(v_2, \mathrm{in}\mathit{Value}(\mathrm{in}\mathit{Set}(vl')))\}, nextv')\})^{+} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{filter}(\mathrm{I}, vl, \mathrm{B}, e', cs, nextv) \\
\quad\quad\quad\quad\quad\quad \mathbf{else}\ \{(e, s_{pre}, s_{post}, cs \cup \{\bot\}, nextv)\} \\
\quad\quad\quad\quad \mathbf{end} \\
\quad\quad \mathbf{end} \\
\;[\!]\ \mathrm{is}\mathit{Call}(\mathrm{I}, lv, v, s'_{pre}, s'_{post})\ \rightarrow\ \mathbf{cases}\ \mathit{accessenv}\ [\![\mathrm{I}]\!]\ e\ \mathbf{of} \\
\quad\quad\quad\quad \mathrm{is}\mathit{Function}(f)\ \rightarrow \\
\quad\quad\quad\quad\quad\quad (\lambda(cs', s''_{post}, nextv').\ \{(e, s_{pre}, s''_{post}, cs', nextv')\})^{+} \\
\quad\quad\quad\quad\quad\quad\quad\quad (f\ \ cons(v,\ \ lv)\ \ cs\ \ s'_{pre}\ \ s'_{post}\ \ v\ \ nextv) \\
\quad\quad\quad\quad \mathbf{else}\ \{(e, s_{pre}, s_{post}, cs \cup \{\bot\}, nextv)\} \\
\quad\quad \mathbf{end} \\
\quad\ \ \vdots \\
\mathbf{end}
\end{array}
$$

Figure 13: More representative cases of function $S$, which defines the effect of simplifying a single constraint. (Part 2 of 2.) Function *filter* is defined in Figure 14.

$filter$ : Identifier $\times$ $List(Storable\text{-}value) \times$ Boolean-expr $\times$ $Environment \times PowerSet(Constraint_{\perp}) \times NextVar$
$\quad \rightarrow PowerSet(List(Storable\text{-}value) \times PowerSet(Constraint_{\perp}) \times NextVar)$

$$
\begin{array}{l}
filter = \lambda(\mathrm{I}, vl, \mathrm{B}, e, cs, nextv). \\
\quad \textbf{if } vl = null \textbf{ then } \{(null, cs, nextv)\} \\
\quad \textbf{else let } e' = updateenv\ [\![I]\!]\ hd(vl)\ e \textbf{ in} \\
\qquad (\lambda(vl', cs', nextv').\ (\lambda(cs'', nextv'').\ \{(cons(hd(vl), vl'), cs'', nextv'')\})^{+} \\
\qquad\qquad\qquad \mathcal{E}[\![\mathrm{B}]\!]\ \mathrm{in}Value(\mathrm{in}Boolean(true))\ (e', cs', nextv') \\
\qquad\qquad \cup\ (\lambda(cs'', nextv'').(\ \{(vl', cs'', nextv'')\})^{+} \\
\qquad\qquad\qquad \mathcal{E}[\![\mathrm{B}]\!]\ \mathrm{in}Value(\mathrm{in}Boolean(false))\ (e', cs', nextv'))^{+} \\
\qquad\quad filter(\mathrm{I}, tl(vl), \mathrm{B}, e, cs, nextv)
\end{array}
$$

Figure 14: Function *filter* is used to select only the elements of a list that satisfy some boolean condition.

In the proof, we work with the following (stronger) formulation of this theorem to make the proof easier. The predicate consistent determines whether or not a set of constraints is consistent (all constraints in the set can be satisfied simultaneously) under a given environment and pair of stores. Thus, 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. If consistent is undefined (returns $\perp$), then the specification contained some type error or was otherwise undefined. As such cases do not produce output (see Figure 4), we simply define the theorem to hold in such cases.

**Lemma 5.2** *Soundness Theorem (stronger version):*

$$
\begin{array}{l}
\forall B \in Boolean\text{-}expr.\ \forall(cs, nextv') \in \mathcal{B}[\![B]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}. \\
\quad \forall(e', s'_{pre}, s'_{post}, cs', nextv'') \in (fix\ \ R^{+})\ \{(e, s_{pre}, s_{post}, cs, nextv')\}. \\
\qquad \textbf{if } consistent(cs, e, s_{pre}, s'_{post}) = \perp \textbf{ then } true \\
\qquad \textbf{else } consistent(cs', e, s_{pre}, s'_{post}) \Leftrightarrow B[e, s_{pre}, s'_{post}]
\end{array}
$$

We define consistent more carefully as follows.

**Definition 5.3** *Definition of consistent*

*A set of constraints is 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.*

If a set of constraints contains $\perp$ and the rest of the constraints in the set are inconsistent, then the value of consistent for that set is false. If a set of constraints contains $\perp$ and the rest of the constraints in the set are consistent, then the value of consistent for that set is undefined.

We use function eval to evaluate one constraint in the given environment and stores, assuming that all logical variables have been labeled. Hence, a set of constraints is consistent iff there is some labeling for which eval is true for each constraint in the set. For most constraints, eval has the same truth value as the result of applying the SPECS-C++ operator associated with the constraint. We provide the definition for three of the more interesting cases (including constraints representing universal quantification and negation) below. Using $v \in ds$ is a slight abuse of notation, since $ds$ is of type *Set* and not a mathematical set. Note that if a set of constraints contains any in*False*() constraint, the set is not consistent.

**Definition 5.4** *Definition of eval*

$$
\begin{array}{l}
eval : Constraint_{\perp} \times Environment \times Store \times Store \rightarrow Boolean_{\perp} \\
eval(c, e, s_{pre}, s_{post}) = \\
\quad \textbf{if } c = \perp \textbf{ then } \perp \\
\quad \textbf{else cases } c \textbf{ of}
\end{array}
$$

$$
\begin{array}{l}
\quad \mathrm{is}\mathit{Forall}(I, ds, B, e', s'_{pre}, s'_{post}) \;\rightarrow \\
\qquad \bigwedge_{v \in ds} B[\mathit{updateenv}\ [\![I]\!]\ v\ e', s'_{pre}, s'_{post}] \\
[\!]\ \mathrm{is}\mathit{Not}(B, e', s'_{pre}, s'_{post}) \;\rightarrow \\
\qquad \mathbf{if}\ B[e', s'_{pre}, s'_{post}]\ \mathbf{then}\ \mathit{false}\ \mathbf{else}\ \mathit{true} \\
[\!]\ \mathrm{is}\mathit{False}() \;\rightarrow\; \mathit{false} \\
\quad \ldots
\end{array}
$$

To show the Soundness Theorem, we use the following lemmas. First, we must show that the valuation function $\mathcal{B}$ is sound.

**Lemma 5.5** *Soundness of* $\mathcal{B}$*:*

$$
\begin{array}{l}
\forall B \in \mathit{Boolean\text{-}expr}.\ \forall(cs, nextv') \in \mathcal{B}[\![B]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}. \\
\quad \forall(e', s'_{pre}, s'_{post}, cs', nextv'') \in (\mathit{fix}\ R^{+})\ \{(e, s_{pre}, s_{post}, cs, nextv')\}. \\
\qquad \mathbf{if}\ \mathit{consistent}(cs, e, s_{pre}, s'_{post}) = \bot\ \mathbf{then}\ \mathit{true} \\
\qquad \mathbf{else}\ \mathit{consistent}(cs, e, s_{pre}, s'_{post}) \Leftrightarrow B[e, s_{pre}, s'_{post}]
\end{array}
$$

That is, that under a valid final post-state store $s'_{post}$, the set of constraints returned by $\mathcal{B}$ is consistent if and only if the assertion B is true. (Note that we use the set of constraints $cs$ returned from $\mathcal{B}$, and not the set of constraints $cs'$ returned from $\mathit{fix}\ R^{+}$, as the argument to consistent.)

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

**Lemma 5.6** *Soundness of* ($\mathit{fix}\ R^{+}$)*:*

$$
\begin{array}{l}
\forall B \in \mathit{Boolean\text{-}expr}.\ \forall(cs, nextv') \in \mathcal{B}[\![B]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}. \\
\quad \forall(e', s'_{pre}, s'_{post}, cs', nextv'') \in (\mathit{fix}\ R^{+})\ \{(e, s_{pre}, s_{post}, cs, nextv')\}. \\
\qquad \mathbf{if}\ \mathit{consistent}(cs, e, s_{pre}, s'_{post}) = \bot\ \mathbf{then}\ \mathit{true} \\
\qquad \mathbf{else}\ \mathit{consistent}(cs, e, s_{pre}, s'_{post}) \Leftrightarrow \mathit{consistent}(cs', e, s_{pre}, s'_{post})
\end{array}
$$

Together, Lemmas 5.5 and 5.6 imply the Soundness Theorem (Lemma 5.2). Note that we are not proving completeness (even though we are using $\Leftrightarrow$), because we do not show that $\mathit{fix}\ R^{+}$ always simplifies a set of constraints to the empty set. In fact, $\mathit{fix}\ R^{+}$ does not always simplify a set of constraints, even when such simplification is possible.

What remains is to prove Lemmas 5.5 and 5.6. We assume that consistent$(cs, e, s_{pre}, s'_{post})$ does not return $\bot$ in these proofs, as the lemmas both hold by definition in that case. This allows us to work with a classical two-valued logic in the proofs.

We prove Lemma 5.5 by induction on the structure of Boolean-expr and the definition of $\mathcal{B}$ given in Figure 8. The base cases are `true`, `false` and the applications of SPECS-C++ relations. For `true`, $\mathcal{B}[\![\mathtt{true}]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post} = \{(\{\}, nextv)\}$. The only element of the constraint set returned is $\{\}$, and so:

$$
\begin{array}{ll}
 & \mathrm{consistent}(cs, e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by } cs = \{\} \rangle \\
 & \mathrm{consistent}(\{\}, e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by the definition of consistent} \rangle \\
 & \mathit{true} \\
\Leftrightarrow & \langle \text{by the definition of SPECS-C++} \rangle \\
 & \mathtt{true}[e, s_{pre}, s'_{post}]
\end{array}
$$

For `false`, $\mathcal{B}[\![\mathtt{false}]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post} = \{(\{\mathrm{in}\mathit{False}()\}, nextv)\}$. Hence:

$$
\begin{array}{ll}
 & \mathrm{consistent}(cs, e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by } cs = \{\mathrm{in}\mathit{False}()\} \rangle \\
 & \mathrm{consistent}(\{\mathrm{in}\mathit{False}()\}, e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by the definition of consistent} \rangle \\
 & \mathit{false}
\end{array}
$$

$\Leftrightarrow$ $\langle$by the definition of SPECS-C++$\rangle$
$\quad \texttt{false}[e, s_{pre}, s'_{post}]$

Applications of SPECS-C++ relations are translated directly into the corresponding constraint. Applying consistent to that constraint is equivalent to evaluating the relation application by definition of eval, assuming the soundness of valuation function $\mathcal{E}$ for syntax domain Expression (see Lemma 5.9).

The inductive step consists of proving the lemma for the boolean connectives and quantifiers.

For negation, we first need to comment on the role of the post-state store that is recorded as part of a *Not* constraint. Because the simplification of a constraint may be delayed for an arbitrarily length of time after the constraint is generated, the current post-state store at the time the constraint is simplified may be different from the post-state store at the time the constraint was generated. In particular, this will occur when a *Not* constraint is generated as part of processing a "test case" (an element S of syntax domain Statement), but is simplified during the processing of a different element of syntax domain Statement. As elements of this domain are statements, a different post-state store results from evaluating each. However, in the case we are discussing here, we are processing one element B of syntax domain Boolean-expr (which is a component of some element of syntax domain Statement). Hence, the current post-state store and the post-state store recorded as part of a *Not* constraint (or any other constraint that has a post-state store component) must be the same. This is true because they are initially the same (the current post-state store is recorded when the constraint is created), and any modifications made to the current post-state store will also be reflected in the post-state stores recorded in constraints. When a *Post* constraint is simplified by $S$ (see Figure 13), an *Equal* constraint is added to the constraint set, and some subsequent execution of $S$ will apply the substitutions resulting from this *Equal* constraint to both the current post-state store and post-state stores contained in other constraints in the same way, and in the same call — see the substitutions done in case is*Equal* of Figure 12. Hence, these stores are always the same (again, for processing one element of domain Boolean-expr). We record this fact as a lemma below.

**Lemma 5.7** *Property of stores contained within constraints:*

*When simplifying constraints obtained from a single element of domain Boolean-expr, any post-state store contained within a constraint is the same as the current (global) post-state store.* ∎

Now, we are ready to prove Lemma 5.5 for *Not* constraints. We have:

$\mathcal{B}[\![\,!\,\mathrm{B}]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post} = \{(\{\mathrm{in}Not(\mathrm{B}, e, s_{pre}, s_{post})\}, nextv)\}$

and hence:

$\quad \mathrm{consistent}(\{\mathrm{in}Not(\mathrm{B}, e, s_{pre}, s_{post})\}, e, s_{pre}, s'_{post})$
$\Leftrightarrow$ $\langle$by Lemma 5.7$\rangle$
$\quad \mathrm{consistent}(\{\mathrm{in}Not(\mathrm{B}, e, s_{pre}, s'_{post})\}, e, s_{pre}, s'_{post})$
$\Leftrightarrow$ $\langle$by the definition of consistent$\rangle$
$\quad \mathrm{eval}(\mathrm{in}Not(\mathrm{B}, e, s_{pre}, s'_{post}), e, s_{pre}, s'_{post})$
$\Leftrightarrow$ $\langle$by the definition of eval$\rangle$
$\quad not\ \mathrm{B}[e, s_{pre}, s'_{post}]$
$\Leftrightarrow$ $\langle$by the definition of `!` in SPECS-C++$\rangle$
$\quad (\,!\,\mathrm{B})[e, s_{pre}, s'_{post}]$

For conjunction,

$\quad \mathcal{B}[\![\mathrm{B}_1 \ /\backslash\ \mathrm{B}_2]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}$
$=$ $\langle$by the definition of $\mathcal{B}\rangle$
$\quad (\lambda(c', nextv').\ (\lambda(c'', nextv'').\ \{(c' \cup c'', nextv'')\})^{+}\ \mathcal{B}[\![\mathrm{B}_2]\!]\ (e, \{\}, nextv')\ s_{pre}\ s_{post})^{+}$
$\qquad \mathcal{B}[\![\mathrm{B}_1]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}$
$=$ $\langle$by the definition of $f^{+}$ for any function $f\rangle$
$\quad (\lambda(c', nextv').\ \bigcup\{\{(c' \cup c'', nextv'')\} \mid (c'', nextv'') \in \mathcal{B}[\![\mathrm{B}_2]\!]\ (e, \{\}, nextv')\ s_{pre}\ s_{post}\})^{+}$
$\qquad \mathcal{B}[\![\mathrm{B}_1]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}$

$$
\begin{array}{ll}
= & \langle \text{by the definition of } \bigcup \rangle \\
 & (\lambda(c', nextv').\ \{(c' \cup c'', nextv'') \mid (c'', nextv'') \in \mathcal{B}[\![\mathrm{B}_2]\!]\ (e, \{\}, nextv')\ s_{pre}\ s_{post}\})^{+} \\
 & \quad \mathcal{B}[\![\mathrm{B}_1]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post} \\
= & \langle \text{by the definition of } f^{+} \text{ for any function } f \rangle \\
 & \bigcup\{\{(c' \cup c'', nextv'') \mid (c'', nextv'') \in \mathcal{B}[\![\mathrm{B}_2]\!]\ (e, \{\}, nextv')\ s_{pre}\ s_{post}\} \\
 & \quad \mid (c', nextv') \in \mathcal{B}[\![\mathrm{B}_1]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}\}
\end{array}
$$

Hence, any constraint set passed to function consistent is of the form $c' \cup c''$, where $c'$ consists of constraints generated from $\mathrm{B}_1$ and $c''$ consists of constraints generated from $\mathrm{B}_2$, and:

$$
\begin{array}{ll}
 & \text{consistent}(cs, e, s_{pre}, s'_{post}) \\
\Rightarrow & \langle \text{by } cs = c' \cup c'' \rangle \\
 & \text{consistent}(c' \cup c'', e, s_{pre}, s'_{post}) \\
\Rightarrow & \langle \text{any subset of a consistent set of constraints is consistent} \rangle \\
 & \text{consistent}(c', e, s_{pre}, s'_{post})\ \wedge\ \text{consistent}(c'', e, s_{pre}, s'_{post}) \\
\Rightarrow & \langle \text{by the inductive hypothesis twice} \rangle \\
 & \mathrm{B}_1[e, s_{pre}, s'_{post}]\ \wedge\ \mathrm{B}_2[e, s_{pre}, s'_{post}] \\
\Rightarrow & \langle \text{by the definition of SPECS-C++} \rangle \\
 & (\mathrm{B}_1\ /\!\backslash\ \mathrm{B}_2)[e, s_{pre}, s'_{post}]
\end{array}
$$

For the only if direction of the proof:

$$
\begin{array}{ll}
 & (\mathrm{B}_1\ /\!\backslash\ \mathrm{B}_2)[e, s_{pre}, s'_{post}] \\
\Rightarrow & \text{by the definition of } /\!\backslash \text{ in SPECS-C++} \\
 & \mathrm{B}_1[e, s_{pre}, s'_{post}]\ \wedge\ \mathrm{B}_2[e, s_{pre}, s'_{post}] \\
\Rightarrow & \langle \text{by tautology} \rangle \\
 & \neg\neg(\mathrm{B}_1[e, s_{pre}, s'_{post}]\ \wedge\ \mathrm{B}_2[e, s_{pre}, s'_{post}]) \\
\Rightarrow & \langle \text{by DeMorgan's Law} \rangle \\
 & \neg(\neg\mathrm{B}_1[e, s_{pre}, s'_{post}]\ \vee\ \neg\mathrm{B}_2[e, s_{pre}, s'_{post}]) \\
\Rightarrow & \langle \text{by the inductive hypothesis twice} \rangle \\
 & \neg(\neg\text{consistent}(c', e, s_{pre}, s'_{post})\ \vee\ \neg\text{consistent}(c'', e, s_{pre}, s'_{post})) \\
\Rightarrow & \langle \text{adding constraints to an inconsistent set cannot make it consistent} \rangle \\
 & \neg(\neg\text{consistent}(c' \cup c'', e, s_{pre}, s'_{post})) \\
\Rightarrow & \langle \text{by } cs = c' \cup c'' \rangle \\
 & \text{consistent}(cs, e, s_{pre}, s'_{post})
\end{array}
$$

For disjunction:

$$
\begin{array}{ll}
 & \mathcal{B}[\![\mathrm{B}_1\ \backslash\!/\ \mathrm{B}_2]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post} \\
= & \langle \text{by the definition of } \mathcal{B} \rangle \\
 & \mathcal{B}[\![\mathrm{B}_1]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}\ \cup \mathcal{B}[\![\mathrm{B}_2]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}
\end{array}
$$

Hence, for any $cs \in \mathcal{B}[\![\ \mathrm{B}_1\ \backslash\!/\ \mathrm{B}_2]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}$, either $cs \in \mathcal{B}[\![\mathrm{B}_1]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}$ or $cs \in \mathcal{B}[\![\mathrm{B}_2]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}$.

Case 1 ($cs \in \mathcal{B}[\![\mathrm{B}_1]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}$):

$$
\begin{array}{ll}
 & \text{consistent}(cs, e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by the inductive hypothesis} \rangle \\
 & \mathrm{B}_1[e, s_{pre}, s'_{post}]
\end{array}
$$

Case 2 ($cs \in \mathcal{B}[\![\mathrm{B}_2]\!]\ (e, \{\}, nextv)\ s_{pre}\ s'_{post}$):

$$
\text{consistent}(cs, e, s_{pre}, s'_{post})
$$

$\Leftrightarrow$ $\langle$by the inductive hypothesis$\rangle$
$B_2[e, s_{pre}, s'_{post}]$

Hence:

consistent$(cs, e, s_{pre}, s'_{post})$
$\Leftrightarrow$ $\langle$by the above and proof by cases$\rangle$
$B_1[e, s_{pre}, s'_{post}] \ \vee\ B_2[e, s_{pre}, s'_{post}]$
$\Leftrightarrow$ $\langle$by the definition of SPECS-C++$\rangle$
$(B_1 \texttt{ \textbackslash/ } B_2)[e, s_{pre}, s'_{post}]$

For implication:

$\mathcal{B}[\![B_1 \texttt{ => } B_2]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}$
$=$ $\langle$by the definition of $\mathcal{B}\rangle$
$\mathcal{B}[\![(B_1 \texttt{ /\textbackslash } B_2) \texttt{ \textbackslash/ !}B_1]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}$

Hence, for any $cs \in \mathcal{B}[\![B_1 \texttt{ => } B_2]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}$:

consistent$(cs, e, s_{pre}, s'_{post})$
$\Leftrightarrow$ $\langle$by previous cases of Lemma 5.5 for /\, \/ and !$\rangle$
$(B_1 \texttt{ /\textbackslash } B_2) \texttt{ \textbackslash/ !}B_1)[e, s_{pre}, s'_{post}]$
$\Leftrightarrow$ $\langle$by tautology $(P \Rightarrow Q) \Leftrightarrow ((P \wedge Q) \vee \neg P)\rangle$
$(B_1 \texttt{ => } B_2)[e, s_{pre}, s'_{post}]$

For universal quantification:

$\mathcal{B}[\![\texttt{\textbackslash forall}\ T\ I\ \texttt{[}(I \texttt{ \textbackslash in } E \texttt{ /\textbackslash } B_1\texttt{)} \texttt{ => } B_2 \texttt{ ]}]\!]\ \ (e, \{\}, nextv)\ s_{pre}\ s_{post}$
$=$ $\langle$by the definition of $\mathcal{B}\rangle$
**let** $(v, nextv') = newvar(nextv)$ **in**
$(\lambda(c', nextv'').\ \{(c' \ \cup\ \{\text{in}Forall(I, v, B_1 \texttt{ => } B_2, e, s_{pre}, s_{post})\}, nextv'')\})^+$
$\mathcal{E}[\![E]\!]\ v\ (e, \{\}, nextv')\ s_{pre}\ s_{post}$
$=$ $\langle$by the definition of $f^+$ for any $f\rangle$
**let** $(v, nextv') = newvar(nextv)$ **in**
$\{(c' \cup \{\text{in}Forall(I, v, B_1 \texttt{ => } B_2, e, s_{pre}, s_{post})\}, nextv') \mid (c', nextv') \in \mathcal{E}[\![E]\!]\ v\ (e, \{\}, nextv)\ s_{pre}\ s_{post}\}$

Hence (using $E[e, s_{pre}, s'_{post}]$ to mean the value of E under the given environment and stores):

consistent$(c' \cup \{\text{in}Forall(I, v, B_1 \texttt{ => } B_2, e, s_{pre}, s_{post})\}, e, s_{pre}, s'_{post})$
$\Leftrightarrow$ $\langle$by Lemma 5.7$\rangle$
consistent$(c' \cup \{\text{in}Forall(I, v, B_1 \texttt{ => } B_2, e, s_{pre}, s'_{post})\}, e, s_{pre}, s'_{post})$
$\Leftrightarrow$ $\langle$by Lemma 5.9 — the soundness of $\mathcal{E}\rangle$
consistent$(\{\text{in}Forall(I, ds, B_1 \texttt{ => } B_2, e, s_{pre}, s'_{post})\}, e, s_{pre}, s'_{post})$, with $ds = E[e, s_{pre}, s'_{post}]$, where E is the domain of the bound variable I from the original assertion (above)
$\Leftrightarrow$ $\langle$by the definition of consistent$\rangle$
eval$(\text{in}Forall(I, ds, B_1 \texttt{ => } B_2, e, s_{pre}, s'_{post}), e, s_{pre}, s'_{post})$
$\Leftrightarrow$ $\langle$by the definition of eval$\rangle$
$\bigwedge_{v' \in ds}(B_1 \texttt{ => } B_2)[updateenv\ [\![I]\!]\ v'\ e, s_{pre}, s'_{post}]$, where $v'$ is a fresh variable
$\Leftrightarrow$ $\langle$by the definition of $\bigwedge\rangle$
$v' \in ds \Rightarrow ((B_1 \texttt{ => } B_2)[updateenv\ [\![I]\!]\ v'\ e, s_{pre}, s'_{post}])$
$\Leftrightarrow$ $\langle$by the definition of $updateenv\rangle$
$(I \texttt{ \textbackslash in } ds \Rightarrow (B_1 \texttt{ => } B_2))[e, s_{pre}, s'_{post}]$
$\Leftrightarrow$ $\langle$by $ds = E[e, s_{pre}, s'_{post}]\rangle$
$(I \texttt{ \textbackslash in } E \texttt{ => } (B_1 \texttt{ => } B_2))[e, s_{pre}, s'_{post}]$
$\Leftrightarrow$ $\langle$by tautology: $(P \Rightarrow (Q \Rightarrow R)) \Leftrightarrow ((P \wedge Q) \Rightarrow R)\rangle$
$((I \texttt{ \textbackslash in } E \texttt{ /\textbackslash } B_1) \texttt{ => } B_2)[e, s_{pre}, s'_{post}]$

$\Leftrightarrow$ $\langle$by generalization$\rangle$
($\forall$ I ((I `\in` E `/\` B$_1$) `=>` B$_2$))$[e, s_{pre}, s'_{post}]$
$\Leftrightarrow$ $\langle$by the definition of SPECS-C++$\rangle$
(`\forall` T I [(I `\in` E `/\` B$_1$) `=>` B$_2$])$[e, s_{pre}, s'_{post}]$

For existential quantification:

$\mathcal{B}[\![$`\exists` T I `[B]`$]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}$
$=$ $\langle$by the definition of $\mathcal{B}\rangle$
**let** $(v, nextv') = newvar(nextv)$ **in**
$\mathcal{B}[\![\mathrm{B}]\!]\ (updateenv\ [\![I]\!]\ v\ e, \{\}, nextv')\ s_{pre}\ s_{post}$

Hence, for any $(cs, nextv'') \in \mathcal{B}[\![$`\exists` T I `[B]`$]\!]\ (e, \{\}, nextv)\ s_{pre}\ s_{post}$:

consistent$(cs, e, s_{pre}, s'_{post})$
$\Leftrightarrow$ $\langle$by the inductive hypothesis$\rangle$
B$[updateenv\ [\![I]\!]\ v\ e, s_{pre}, s'_{post}]$
$\Leftrightarrow$ $\langle$by the definition of *updateenv* and tautology: $P(v) \Leftrightarrow \exists x P(x))\rangle$
$\exists$ I B$[e, s_{pre}, s'_{post}]$
$\Leftrightarrow$ $\langle$by the definition of SPECS-C++$\rangle$
`\exists` T I `[B]`$[e, s_{pre}, s'_{post}]$

This completes the proof of Lemma 5.5. ∎

To prove Lemma 5.6, we note that $R$ simply applies function $S$ repeatedly (in the fixed point construction) to simplify a set of constraints (see Figure 10). Hence, we prove Lemma 5.6 by induction on the number of applications of $S$.

For the basis, we define $S^0$ (0 applications of $S$) to be the identity function, and:

$$\text{consistent}(cs, e, s_{pre}, s'_{post}, nextv) \Leftrightarrow \text{consistent}(cs, e, s_{pre}, s'_{post}, nextv)$$

is a tautology.

For the inductive step, we need to look at the sequence of applications of $S$ that produces the final constraint set $cs'$ and final store $s'_{post}$. We state this step as a lemma.

**Lemma 5.8** *Soundness of* $S$*:*

*At some arbitrary point in a sequence of applications of* $S$*, let* $cs''$ *be the current constraint set and let* $c$ *be the constraint chosen for simplification. Furthermore, let* $cs'''$ *be any of the constraint sets returned by* $S$*. That is:*

$$(e'', s''_{pre}, s''_{post}, cs''', nextv'') \in S\ c\ (e, s_{pre}, s_{post}, cs'' - \{c\}, nextv')$$

*then:*

$$consistent(cs'', e, s_{pre}, s'_{post}) \Leftrightarrow consistent(cs''', e, s_{pre}, s'_{post})$$

Lemma 5.8 captures precisely how $S$ is used by $R$. We prove Lemma 5.8 by showing that it is true for each kind of constraint, i.e. for each possible value of $c$.

For *Forall* constraints (representing universal quantification), we have from the definition of $S$ (Figure 12), with some renaming of variables for clarity:

$S$ in*Forall*(I$, ds, $B$', e', s_{pre}, s_{post})(e, s_{pre}, s''_{post}, cs'' - \{$in*Forall*(I$, ds, $B$', e', s_{pre}, s_{post})\}, nextv)$
$=$ $\langle$by Lemma 5.7$\rangle$
$S$ in*Forall*(I$, ds, $B$', e', s_{pre}, s''_{post})(e, s_{pre}, s''_{post}, cs'' - \{$in*Forall*(I$, ds, $B$', e', s_{pre}, s''_{post})\}, nextv)$
$=$ $\langle$by the definition of $S\rangle$
**cases** $ds$ **of**

$$
\begin{array}{l}
\quad\quad \text{is}\mathit{Value}(d) \;\rightarrow\; \textbf{cases } d \textbf{ of} \\
\quad\quad\quad\quad \text{is}\mathit{Set}(vl) \;\rightarrow\; (\textbf{if } vl = nil \textbf{ then } \{(e, s_{pre}, s''_{post}, cs'', nextv)\} \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else let } (v, nextv') = newvar(nextv) \textbf{ in} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{let } e'' = updateenv\; [\![\mathrm{I}]\!]\; v\; e' \;\; \textbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{let } (v', nextv'') = newvar(nextv') \textbf{ in} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\lambda(cs', nextv''').\{(e, s_{pre}, s''_{post}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad cs' \cup \{\text{in}\mathit{Member}(v, ds), \text{in}\mathit{Minus}(ds, v, v'), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{in}\mathit{Forall}(\mathrm{I}, v', \mathrm{B}', e', s_{pre}, s''_{post})\}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad nextv''')\})^{+} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathcal{B}[\![\mathrm{B}']\!]\; (e'', cs'', nextv'')\; s_{pre}\; s''_{post} \\
\quad\quad\quad\quad \textbf{else } \{(e, s_{pre}, s''_{post}, cs'' \cup \{\perp\}, nextv)\} \\
\quad\quad \textbf{end} \\
\quad\; [\!]\; \text{is}\mathit{Variable}(i) \;\rightarrow\; \{(e, s_{pre}, s''_{post}, cs'' \cup \{\text{in}\mathit{Forall}(\mathrm{I}, ds, \mathrm{B}', e', s_{pre}, s''_{post})\}, nextv)\} \\
\quad \textbf{end}
\end{array}
$$

If $ds$ is a variable, we have that $cs''' = cs''$ where $cs'''$ is the constraint set component of the value returned from $S$, and so:

$$\text{consistent}(cs'', e, s_{pre}, s'_{post}) \Leftrightarrow \text{consistent}(cs''', e, s_{pre}, s'_{post}).$$

If $ds$ is a value but not a set, the original constraint set $cs$ did not have a defined value, and so Lemma 5.6 holds by definition.

If $ds$ is the empty set, we have the following:

$$
\begin{array}{l}
cs''' = cs'' - \{\text{in}\mathit{Forall}(\mathrm{I}, ds, \mathrm{B}', e', s_{pre}, s''_{post})\} \\
\text{eval}(\text{in}\mathit{Forall}(\mathrm{I}, ds, \mathrm{B}', e', s_{pre}, s''_{post}), e, s_{pre}, s'_{post}) = true
\end{array}
$$

Hence:

$$
\begin{array}{ll}
 & \text{consistent}(cs''', e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by eval}(\text{in}\mathit{Forall}(\mathrm{I}, ds, \mathrm{B}', e', s_{pre}, s''_{post}), e, s_{pre}, s'_{post}) = true \text{ and the definition of consistent} \rangle \\
 & \text{consistent}(cs'', e, s_{pre}, s'_{post})
\end{array}
$$

For $ds$ a nonempty set, we have:

$$
\begin{array}{l}
cs''' = (cs'' - \{\text{in}\mathit{Forall}(\mathrm{I}, ds, \mathrm{B}', e', s_{pre}, s''_{post})\}) \cup \\
\quad\quad \{\text{in}\mathit{Forall}(\mathrm{I}, v', \mathrm{B}', e', s_{pre}, s''_{post}), \text{in}\mathit{Member}(v, ds), \text{in}\mathit{Minus}(ds, v, v')\}
\end{array}
$$

Hence:

$$
\begin{array}{ll}
 & \text{consistent}(cs''', e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by the equality given above} \rangle \\
 & \text{consistent}((cs'' - \{\text{in}\mathit{Forall}(\mathrm{I}, ds, \mathrm{B}', e', s_{pre}, s''_{post})\}) \cup \\
 & \quad\quad \{\text{in}\mathit{Forall}(\mathrm{I}, v', \mathrm{B}', e', s_{pre}, s''_{post}), \text{in}\mathit{Member}(v, ds), \text{in}\mathit{Minus}(ds, v, v')\}, e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by tautology } ((\forall x . x \in S \Rightarrow P(x)) \wedge P(v)) \Leftrightarrow (\forall x . x \in S \cup \{v\} \Rightarrow P(x)) \text{ and the definition of consistent} \rangle \\
 & \text{consistent}(cs'', e, s_{pre}, s'_{post})
\end{array}
$$

For $Not$ constraints (representing negation), we have from the definition of $S$ (Figure 12), with some renaming of variables for consistency:

$$
\begin{array}{ll}
 & S\ \text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s_{post})\ (e, s_{pre}, s''_{post}, cs'' - \{\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s_{post})\}, nextv) \\
= & \langle \text{by Lemma 5.7} \rangle \\
 & S\ \text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s''_{post})\ (e, s_{pre}, s''_{post}, cs'' - \{\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s''_{post})\}, nextv) \\
= & \langle \text{by the definition of } S \rangle \\
 & \textbf{let } res = (\mathit{fix}\ R^{+})\ (adjustState\ e'\ s_{pre}\ s''_{post} \\
 & \quad\quad\quad\quad\quad\quad (\mathcal{B}[\![\mathrm{B}']\!]\ (e', cs'' - \{\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s''_{post})\}, nextv)\ s_{pre}\ s''_{post}))\ \textbf{in}
\end{array}
$$

**if** $\exists(e'', s'_{pre}, s'''_{post}, cs'''', nextv') \in res \,.\, cs'''' = \{\}$ **then**
$\{(e, s_{pre}, s''_{post}, \{\text{in}\mathit{False}()\}, nextv')\}$
**else if** $\forall(e', s''_{pre}, s''_{post}, cs', nextv') \in res\,.\text{in}\mathit{False}() \in cs'''$ **then**
$\{(e, s_{pre}, s''_{post}, cs'' - \{\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s''_{post})\}, nextv')\}$
**else** $\{(e, s_{pre}, s''_{post}, cs \cup \{\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s''_{post})\}, nextv)\}$

We inductively assume that Lemma 5.2 holds for the nested use of *fix* $R^+$ above. That is:

$$\forall(e'', s'_{pre}, s'''_{post}, cs'''', nextv') \in res \,.\, \text{consistent}(cs'''', e, s_{pre}, s'_{post}) \Leftrightarrow \mathrm{B}'[e, s_{pre}, s'_{post}]$$

If $\exists(e'', s'_{pre}, s'''_{post}, cs'''', nextv') \in res \,.\, cs'''' = \{\}$, then we have that $cs''' = \{\text{in}\mathit{False}()\}$. To show this case, we first show that the *Not* constraint involved evaluates to false. We can then show that neither $cs''$ nor $cs'''$ is consistent, and so are equivalent.

First:

$$\begin{array}{ll}
 & \text{eval}(\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s_{post}), e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by Lemma 5.7} \rangle \\
 & \text{eval}(\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s'_{post}), e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by the definition of eval} \rangle \\
 & \mathit{not}\ \mathrm{B}'[e', s_{pre}, s'_{post}] \\
\Leftrightarrow & \langle \text{by inductive use of the Soundness Theorem} \rangle \\
 & \mathit{not}\ \text{consistent}(cs'''', e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by } cs'''' = \{\} \text{ and the definition of consistent} \rangle \\
 & \mathit{not}\ \mathit{true} \\
\Leftrightarrow & \\
 & \mathit{false}
\end{array}$$

Now:

$$\begin{array}{ll}
 & \text{consistent}(cs'', e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by } cs'' = \{\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s_{post})\} \cup (cs'' - \{\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s_{post})\}) \rangle \\
 & \text{consistent}(\{\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s_{post})\} \cup (cs'' - \{\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s_{post})\}), e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by the above and the definition of consistent} \rangle \\
 & \mathit{false} \\
\Leftrightarrow & \langle \text{by the definition of consistent} \rangle \\
 & \text{consistent}(\{\text{in}\mathit{False}()\}, e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by } cs''' = \{\text{in}\mathit{False}()\} \rangle \\
 & \text{consistent}(cs''', e, s_{pre}, s'_{post})
\end{array}$$

If $\forall(e'', s'_{pre}, s'''_{post}, cs'''', nextv') \in res \,.\, \text{in}\mathit{False}() \in cs''''$, we have that $cs'' = cs''' \cup \{\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s_{post})\}$, and we can also show that $\text{eval}(\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s_{post}), e, s_{pre}, s'_{post})$ is true as follows:

$$\begin{array}{ll}
 & \text{eval}(\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s_{post}), e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by Lemma 5.7} \rangle \\
 & \text{eval}(\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s'_{post}), e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by the definition of eval} \rangle \\
 & \mathit{not}\ \mathrm{B}'[e', s_{pre}, s'_{post}] \\
\Leftrightarrow & \langle \text{by inductive use of the Soundness Theorem} \rangle \\
 & \mathit{not}\ \text{consistent}(cs'''', e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by in}\mathit{False}() \in cs'''' \text{ and the definition of consistent} \rangle \\
 & \mathit{not}\ \ \mathit{false} \\
\Leftrightarrow & \\
 & \mathit{true}
\end{array}$$

Hence:

$$
\begin{array}{ll}
 & \text{consistent}(cs'', e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by } cs'' = cs''' \cup \{\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s'_{post})\} \text{ and eval}(\text{in}\mathit{Not}(\mathrm{B}', e', s_{pre}, s'_{post}), e, s_{pre}, s'_{post}) = \mathit{true}\rangle \\
 & \text{consistent}(cs''', e, s_{pre}, s'_{post})
\end{array}
$$

Finally, if neither $\forall (e'', s'_{pre}, s'''_{post}, cs'''', nextv') \in res\ .\ cs'''' = \{\}$ nor $\forall (e'', s'_{pre}, s'''_{post}, cs'''', nextv') \in res\ .\ \text{in}\mathit{False}() \in cs''''$, is true, then we have that $cs''' = cs''$. There are two cases. If $\text{consistent}(cs'', e, s_{pre}, s'_{post}) = \bot$, then the original set of constraints $cs$ did not have a defined value, and so Lemma 5.6 holds by definition. Otherwise:

$$
\begin{array}{ll}
 & \text{consistent}(cs''', e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by } cs''' = cs''\rangle \\
 & \text{consistent}(cs'', e, s_{pre}, s'_{post})
\end{array}
$$

For *False* constraints (used to indicate failure), we have from the definition of $S$ (Figure 12), with some renaming of variables for consistency:

$$S\ \text{in}\mathit{False}()(e, s_{pre}, s_{post}, cs'' - \{\text{in}\mathit{False}()\}, nextv) = \{(e, s_{pre}, s_{post}, \{\text{in}\mathit{False}()\}, nextv)\}$$

Since $\text{in}\mathit{False}() \in cs''$ and $cs''' = \{\text{in}\mathit{False}()\}$, we have:

$$
\begin{array}{ll}
 & \text{consistent}(cs''', e, s_{pre}, s'_{post}) \\
\Leftrightarrow & \langle \text{by } cs''' = \{\text{in}\mathit{False}()\} \text{ and the definition of consistent}\rangle \\
 & \mathit{false} \\
\Leftrightarrow & \langle \text{by in}\mathit{False}() \in cs'' \text{ and the definition of consistent}\rangle \\
 & \text{consistent}(cs'', e, s_{pre}, s'_{post})
\end{array}
$$

The proof of Lemma 5.8 for the remaining types of constraints is omitted. This completes the proof of Lemma 5.6. ∎

To complete the proof of Lemma 5.2, we need to show the soundness of $\mathcal{E}$.

**Lemma 5.9** *Soundness of* $\mathcal{E}$*:*

$$
\begin{array}{l}
\forall E \in \mathit{Expression}.\ \forall (cs, nextv') \in \mathcal{E}[\![E]\!]\ vl\ (e, \{\}, nextv)\ s_{pre}\ s_{post}. \\
\quad \forall (e', s'_{pre}, s'_{post}, cs', nextv'') \in (\mathit{fix}\ \ R^{+})\ \{(e, s_{pre}, s_{post}, cs, nextv')\}. \\
\qquad \textbf{if } \mathit{consistent}(cs, e, s_{pre}, s'_{post}) = \bot \textbf{ then } \mathit{true} \\
\qquad \textbf{else } \mathit{consistent}(cs', e, s_{pre}, s'_{post}) \Leftrightarrow vl = E[e, s_{pre}, s'_{post}]
\end{array}
$$

Note that the argument $vl$ to $\mathcal{E}$ represents the value of the expression.

To prove the soundness of $\mathcal{E}$, we use two lemmas. These lemmas play the same role in this proof as Lemmas 5.5 and 5.6 in the proof of the Soundness Theorem.

**Lemma 5.10**

$$
\begin{array}{l}
\forall E \in \mathit{Expression}.\ \forall (cs, nextv') \in \mathcal{E}[\![E]\!]\ vl\ (e, \{\}, nextv)\ s_{pre}\ s_{post}. \\
\quad \forall st \in (\mathit{fix}\ \ R^{+})\ \{(e, s_{pre}, s_{post}, cs, nextv')\}. \\
\qquad \textbf{if } \mathit{consistent}(cs, e, s_{pre}, s'_{post}) = \bot \textbf{ then } \mathit{true} \\
\qquad \textbf{else let } (e', s'_{pre}, s'_{post}, cs', nextv'') = st \textbf{ in} \\
\qquad\quad \mathit{consistent}(cs, e, s_{pre}, s'_{post}) \Leftrightarrow vl = E[e, s_{pre}, s'_{post}]
\end{array}
$$

∎

**Lemma 5.11**

$$\begin{array}{l}\forall E \in \mathit{Expression}.\ \forall (cs, nextv') \in \mathcal{E}[\![E]\!]\ vl\ (e, \{\}, nextv)\ s_{pre}\ s'_{post}. \\ \quad \forall (e', s'_{pre}, s'_{post}, cs', nextv'') \in (\mathit{fix}\ R^{+})\ \{(e, s_{pre}, s_{post}, cs, nextv')\}. \\ \qquad \textbf{if}\ \mathit{consistent}(cs, e, s_{pre}, s'_{post}) = \bot\ \textbf{then}\ \mathit{true} \\ \qquad \textbf{else}\ \mathit{consistent}(cs, e, s_{pre}, s'_{post}) \Leftrightarrow \mathit{consistent}(cs', e, s_{pre}, s'_{post})\end{array}$$

∎

Together, these two lemmas imply Lemma 5.9. The proof of Lemma 5.10 (omitted) is accomplished by induction on the structure of syntax domain Expression, and so is very similar to the proof of Lemma 5.5. The proof of Lemma 5.11 (omitted) is accomplished by induction on the number of of applications of $S$, and so is similar to the proof of Lemma 5.6. ∎

# 6 Conclusion

We have presented an algorithm for executing formal specifications by translation to concurrent constraint programs, and we have shown the soundness of that algorithm. This work has two important benefits. First, a specifier can use our algorithm for executing and testing specifications with confidence that all results returned will be correct. Second, by presenting our algorithm in a way that is relatively independent of programming language (or even paradigm), we have made it much easier for others to implement similar execution systems for specification languages such as VDM and Z. Hence, we expect this work to be helpful in expanding the use of formal specifications.

# References


[A+93] D.J. Andrews et al. Information Technology Programming Languages – VDM-SL: First Committee Draft Standard CD1387-1. Document ISO/IEC JTC1/SC22/WG19 N-20, International Standards Organization, November 1993. ftp://gatekeeper.dec.com/pub/standards/vdmsl/.

[BCJ84] H. Barringer, J. H. Cheng, and C. B. Jones. A Logic Covering Undefinedness in Program Proofs. *Acta Informatica*, 21:251 – 269, 1984.

[BMR95] Alex Borgida, John Mylopoulos, and Rayomnd Reiter. On the Frame Problem in Procedure Specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[ELL94] René Elmstrøm, Peter Gorm Larsen, and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, 29(9), September 1994.

[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.

[Hay93] I. Hayes, editor. *Specification Case Studies*. International Series in Computer Science. Prentice-Hall, Englewood Cliffs, N.J., second edition, 1993.

[HJ89] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):320–338, November 1989.

[JH94] Sverker Janson and Seif Haridi. An introduction to AKL - A multiparadigm programming language. In *Constraint Programming (NATO-ASI Series vol. 131)*. Springer Verlag, 1994.

[JM94] C. B. Jones and C. A. Middelburg. A Typed Logic of Partial Functions Reconstructed Classically. *Acta Informatica*, 31(5):399 – 430, 1994.

[JMMS98] Joxan Jaffar, Michael Maher, Kim Marriot, and Peter Stuckey. The Semantics of Constraint Logic Programs. *The Journal of Logic Programming*, 37(1 – 3):1 – 46, 1998.

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

[LP98] Gary T. Leavens and Don Pigozzi. Class-Based and Algebraic Models of Objects. In Rance Cleaveland, Michael Mislove, and Philip Mulry, editors, *US - Brazil Joint Workshops on the Formal Foundations of Software Systems*, volume 14 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1998.

[NF89] Tim Nicholson and Norman Foo. A Denotational Semantics for Prolog. *ACM Transactions on Programming Languages and Systems*, 11(4):650–665, October 1989.

[Sch86] David A. Schmidt. *Denotational Semantics — A Methodology for Language Development*. Wm. C. Brown Publishers, Dubuque, Iowa, 1986.

[Spi89] J. M. Spivey. An Introduction to Z and Formal Specifications. *Software Engineering Journal*, pages 40 – 50, 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.

[SRP91] Vijay Saraswat, Martin C. Rinard, and Prakash Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of the Eighteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 333 – 352, Orlando, Florida, January 1991.

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

[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.

[Win87] Jeannette M. Wing. Writing Larch Interface Language Specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[WLB00] Tim Wahls, Gary T. Leavens, and Albert L. Baker. Executing Formal Specifications with Concurrent Constraint Programming. *Automated Software Engineering*, 7(4), 2000.