HELPING SPECIFIERS EVALUATE THEIR SPECIFICATIONS

CRITÈRES PERMETTANT AUX SPÉCIFIEURS D'ÉVALUER LEURS SPÉCIFICATIONS

JEANNETTE M. WING[1]

Abstract: Current research in specifications aims to put formal specifications to productive use throughout the entire programming process, and especially in program design. In the incremental development of a specification, one problem faced by a specifier is the lack of useful feedback informing him whether the specification is any good. One way of providing such feedback is to supply the specifier with ways to evaluate a specification, e.g., check that some desired property of the specification holds. In the context of a *two-tiered approach* to specifying programs, this paper defines and discusses four properties a specifier may wish to check of his specification, and how he can use the feedback gained from having checked for them. These properties are related to the consistency and completeness of a specification. All definitions are in terms of the theories associated with specifications.

Keywords: Formal specifications, program design, specification methods, specification languages, specification evaluation, specification properties, consistency, completeness, two-tiered approach.

Resume: Un but essentiel de la recherche en matière de spécifications (de programmes) est actuellement d'utiliser les spécifications de manière productive d'un bout à l'autre de l'effort de programation, et notamment dans le cadre de la conception. Au cours des étapes du développement d'un ensemble de spécifications, un des problèmes du spécifieur est le manque de "feedback" destiné à le renseigner sur la qualité de ses spécifications. Une possibilité pour pallier à cette carrence est de fournir au spécifieur des critères moyens d'évaluation pour spécifications (par exemple, verifier que les spécifications ont une certaine propriété). Dans le contexte de la méthode de spécification dite "two-tiered" ("à-deux-niveaux"), cet article definit et examine quatre propriétés qu'un spécifieur est susceptible de vouloir vérifier pour ses spécifications, et comment il peut utiliser le "feedback" obtenu par cette vérification. Ces propriétés sont intimement liées à la consistence et à la complétude des spécifications. Toutes les définitions sont données en termes des théories associées aux spécifications.

Mots cles: Spécifications formelles, conception des programmes, méthodes de spécification, langages de spécifications, évaluation de spécifications, propriétés des spécifications, consistence, complétude, approche à-deux-niveaux.

[1]Computer Science Department, Henry Salvatori Computer Science Center, University of Southern California, Los Angeles, California 90089-0782, U.S.A., (213) 743-7842.

1. Motivation

The broad goal of our research in specifications is to put formal specifications to productive use in the overall programming process. Some of our premises and goals are stated in [Guttag 82]. One of the problems specifiers face when developing a formal specification early in this process, i.e., the design phase, is that no feedback is provided to indicate whether the specification is on the "right track." A specifier may wish to know if his specification is in some sense "correct," i.e., that it captures his intuition of what he is trying to specify, or that it is in some sense "good," i.e., that is satisfies a set of desired objective and possibly subjective properties. Providing such feedback to a specifier can increase his confidence in his specification. It may also help him better understand both the specification and the client's problem statement, and consequently may cause him to modify or improve the specification or force the client to clarify the problem statement. Depending on how informative the feedback is, it may even point to a place in the specification where an improvement can be made.

One way of providing such feedback is to provide the specifier with "checkers," each of which tests for a property of a specification. For example, a syntax-checker would check for the syntactic legality of a specification. In this paper, we consider four properties of specifications related to the consistency and completeness of specifications, and we discuss what a specifier might gain from checking a specification for these properties. We do not give an extensive enumeration of properties, but just a sample to suggest that checking for them would be useful in evaluating specifications. We leave for further work the tasks of identifying and defining additional properties, analyzing the tradeoffs among them, and finding other ways of evaluating specifications.

The ideas underlying the properties of specifications that we define are independent of any particular specification method or language. We find it useful, however, to illustrate them in the context of a *two-tiered approach* to specifying programs in order to be concrete in our examples and definitions. Hence, in Section 2 we give an overview of the two-tiered approach, a specification language that supports this approach, and what the theory of a two-tiered specification is. The results of this paper are taken from [Wing 83], which contains further discussion and details of the approach, the language, and the theories of two-tiered specifications.

In Section 3 we present the definitions of four properties of specifications and discuss how a specifier might use the knowledge gained from checking a specification for these properties. In Section 4 we briefly mention two directions of further work.

2. Overview of Two-Tiered Specifications

2.1. The Two-Tiered Approach

The two-tiered approach to specifying programs separates the specification of underlying abstractions from the specification of state transformations. We use a *shared* specification language to describe underlying abstractions, and an *interface* specification language to describe state transformations. The specification of a program module is written in an interface language and consists of two parts: a *shared language component* and an *interface language component*. These two components correspond to the two tiers in our approach.

The interface specification language is programming language dependent, while the shared language is programming language independent. This allows us to keep separate the description of programming language independent issues from the description of programming language dependent ones, e.g., side effects, exceptional termination, and resource allocation. For example, if we were to implement arithmetic, we would describe ideal arithmetic in the shared language, and we would describe boundary conditions constrained by word and memory size in an interface language.

2.2. A Two-Tiered Specification Language

We describe the specification language used in our examples by first discussing a shared language, Larch [Guttag 83], in Section 2.2.1, and then an interface language for the programming language CLU [Liskov 81] in Section 2.2.2.

2.2.1. A Shared Language

The shared specification language we use is Larch, which evolved from the algebraic style of specification languages [Burstall 81, Nakajima 80, ADJ 75, Guttag 77]. Since Larch's unit of encapsulation is called a *trait*, we will henceforth use "trait" for the term "shared language component." We describe only the syntax and semantics of traits necessary to understand the examples and definitions of this paper.

A trait introduces a set of *sort* identifiers, and a set of *function* identifiers and signatures. The function identifiers are composed to build *terms*, which are used to write assertions appearing in the body of an interface language component. The sort identifiers and function signatures are used to *sort-check* terms much in the same way as type identifiers are used to type-check programs. The meaning of a trait is based on a first-order theory of equality for terms. Appendix I contains an example of a trait, SetOfInt, useful for defining values of sets.

2.2.2. An Interface Language

The interface specification language we describe is for the programming language CLU. A CLU program consists of a set of modules, each of which is either a *procedure* or *cluster*.[2] Hence, the interface language components we will discuss are called *procedure specifications* and *cluster specifications*. A procedure performs an action on a set of objects, and terminates returning a set of objects. A cluster names a *type* and defines a set of procedures that create and manipulate objects of that type. An object has not only a type, but also a *value*. An object whose value can change is said to be *mutable*. A type is mutable if objects of that type are mutable. For example, arrays are mutable, but integers are immutable in CLU.

The key connection between the shared language and CLU is that (sorted) terms denote values of (typed) objects. Interface language components, i.e., procedure and cluster specifications, establish this connection.

A procedure specification has three parts: a header, a link to a trait, and a body. Figure 1 gives an example of a procedure specification. Its header indicates that the input argument is of type set, and the output argument is of type int. Its link is the name of a trait, SetOfInt, which we call the *used trait* of choose. Its body contains two first-order *assertions* that correspond to a pre-condition on the state when the procedure is invoked, i.e., the *initial* state, and a post-condition on both the initial state and the state when the procedure terminates, i.e., the *final* state. For example, choose's pre-condition is satisfied if the initial value of the input argument is not empty. We denote the initial value of an object by an object identifier, e.g., s, followed by an up-arrow (↑); the final value, an object identifier followed by a down-arrow (↓).

[2] We will ignore CLU's third kind of program module called an *iterator*.

```
choose = proc (s: set) returns (i: int)
   uses SetOfInt
      pre ~isEmpty(s↑)
      post has(s↑,i↓) ∧ s↓ = remove(s↑,i↓) ∧ mutates s
   end
```

Figure 1: A Choose Procedure Specification

Assertions in the pre- and post-conditions are constructed from function identifiers, e.g., isEmpty, has, and remove, which are provided by the used trait, plus special assertions, e.g., **mutates** s. Special assertions are introduced to highlight issues particular to the target programming language. The **mutates** assertion in the example states that the choose procedure may mutate no object other than the input set object. The meanings of all assertions are based on the theory of equality defined by the used trait; this provides the link between the two tiers of a specification.

A cluster specification consists of a header, a link to a trait, and a body. Figure 2 gives an example of a mutable set cluster specification, which we call *SetCluSpec* for future reference. Its header names the type, set, which we call the *defined type*, and contains a list of procedure identifiers; its body consists of a set of procedure specifications. Singleton and union are specified to return new nonempty set objects. Delete might mutate its input set argument, if doing so does not empty it; otherwise, it terminates exceptionally, signaling emptiesSet. Size returns the cardinality of its set argument. *SetCluSpec*'s link is given by a **uses** clause, which names the used trait (SetOfInt), and a **provides** clause. SetOfInt supplies all function identifiers that appear in the assertions of set's procedure specifications. The **provides** clause gives a mapping from a type identifier, e.g., set, to a sort identifier, e.g., SI, which must appear in the used trait. This mapping determines the values over which objects of the type defined by the cluster can range. All objects of the type set are restricted to values denoted by terms of sort SI. The **provides** clause also indicates that set is a mutable type via the keyword **mutable**.

```
set = cluster is singleton, union, delete, size
   uses SetOfInt
   provides mutable set from SI

      singleton = proc (i: int) returns (s: set)
         pre true
         post s↓ = add(empty,i↑) ∧ new s
         end

      union = proc (s1, s2: set) returns (s3: set)
         pre true
         post ∀j:E [has(s2↓,j) = has(s1↑,j) ∨ has(s2↑,j)] ∧ new s3
         end

      delete = proc (s: set, i: int) signals (emptiesSet)
         pre card(s↑) ≥ 2 ∨ ~has(s↑,i↑)
         post s↓ = remove(s↑,i↑) ∧ mutates s

         pre card(s↑) eq 1 ∧ has(s↑,i↑)
         post signals emptiesSet ∧ new ∅
         end

      size = proc (s: set) returns (i: int)
         pre true
         post i↓ = card(s↑) ∧ new ∅
         end

   end
```

Figure 2: A Set Cluster Specification (SetClusSpec)

The **signals** and **new** assertions found in the bodies of set's procedure specifications are examples of special assertions introduced to highlight CLU features. We allow the use of multiple pre/post pairs in the body of a procedure specification, e.g., as in delete, to aid in readability. The appearance of i pre-conditions translates to the single pre-condition that is the disjunction of all i pre-conditions; of i post-conditions, to the single post-condition that is the conjunction of i "$pre_i \Rightarrow post_i$" implications.

2.3. Theory of a Specification

We view a specification as a formal system that specifies a theory, where a *formal system*, consists of a language (a set of symbols and a set of well-formed formulae), a set of axioms, and a set of rules of inference. A *theory* specified by a formal system, Spec, is the smallest set of formulae reflexively and transitively closed over the set of axioms under the rules of Spec.

The theory of a trait derives from first-order predicate logic with equations as atomic formulae.[3] The theory of a procedure specification, Pr, extends the theory of its used trait by adding a set of triples, P{Pr}Q, where P is a first-order assertion on the initial values of objects, and Q is a first-order assertion on the initial and final values of objects.

[3] "Predicate" refers to quantification; "first-order" refers to quantification over variables in the atomic formulae.

The theory of a cluster specification is the union of the theories of its procedure specifications, plus the set of triples derivable from a type induction rule associated with a cluster specification. The triples in this latter set are each of the form true{S}Q, where S is any program statement and Q is an assertion on the final state. There is a hypothesis in the rule for each procedure specified to create or mutate an object of that type. The type induction rule lets us associate type invariants with a cluster specification. For example, the type induction rule associated with *SetCluSpec* has three hypotheses, one each for singleton, union, and delete. One of the invariants we can derive from this rule is that all sets are of size strictly greater than zero, i.e., no set objects can be empty. In subsequent examples, we will refer to this (atypical) cardinality property as *SizeInv*.

We denote the theory of a procedure specification, Pr, by Th(Pr). We call a procedure specification found in the body of a cluster specification *bound* and one found outside all cluster specifications *free*. For example, choose (of Figure 1) is free with respect to *SetCluSpec*, but singleton (of Figure 2) is bound. If Pr is bound, then we write Th(Pr+) to denote the Th(Pr) plus the theory of the defined type. Hence, if Pr is bound, triples derived from the type induction rule are included in Th(Pr+), but not in Th(Pr). In our definitions, we write "Pr.pre" for the assertion found in Pr's pre-condition; "Pr.post," in Pr's post-condition.

We focus on expressing our definitions in terms of theories, and not of models, because a theory can be viewed as a set of formulae, i.e., strings of symbols, subject to syntactic manipulation. Given sufficient mechanized theorem-proving power, we can reason about specifications independently of their models by manipulating their theories just as we do their text. Therefore, our definitions of properties will be in terms of whether a formula is or is not in a theory, i.e., whether it is or is not deducible from the specification.

3. Properties of a Specification

Following our specification approach, we put together pieces of existing specifications to create a larger specification targeted for a particular problem or problem domain. As the specification grows incrementally, we might invoke a "checker" to test for a property of the specification. In the process of tuning a specification, we would probably invoke such a checker many times. If a specification does not have a property, we can choose either to modify the specification so that it does, or accept the fact that it does not--a checker is used only to provide information, not to inhibit the progress of writing the specification. Checking for a property might also necessitate a clarification in the client's problem statement. For example, discovering that a specification is inconsistent may point to a contradiction in the problem statement--the specification merely reflected the mistake.

Two properties of a specification that might be of interest are consistency and completeness. The ability to check for consistency is probably of more use than the ability to check for completeness. Knowing a specification is inconsistent informs the specifier that no implementation could be written to satisfy the specification. We define consistency in Section 3.1.

We do not define completeness because we expect most specifications to be incomplete in the logical sense[4] as well as in the practical sense--in the development of a large specification, we may have no intention of ever "finishing" it. We usually want to know when we have said "enough" as opposed to "everything." In Sections 3.2-3.4 we define three properties: full-coverage, determinism, and protection. Each gets at a different notion of sufficiency as a different kind of approximation to completeness. For each property we first motivate it, then define it, and then discuss specifications

[4]Given a formal system, its theory is *complete* if for all formulae, F, we can determine whether F or ~F is in the theory.

with that property.

3.1. Consistency

The usual notion of consistency of a formal system refers to the inability to derive an explicit contradiction. For a given first-order predicate logic formal system, a set of formulae, ϕ, is *inconsistent* if and only if false is in ϕ. We use this definition to build the notion of an inconsistent specification.

Def: A trait, Tr, is *inconsistent* if and only if the formula (true = false) or the formula false is in Th(Tr).

Def: A procedure specification, Pr, is *inconsistent* if and only if (1) there exists a satisfiable formula P such that the formula P{Pr}false is in Th(Pr), or (2) Pr's used trait is inconsistent.

Def: A cluster specification, Cl, is *inconsistent* if and only if (1) true{S}false is in Th(Cl), or (2) for any of Cl's procedure specifications, Pr, there exists a satisfiable formula P such that the formula P{Pr}false is in Th(Cl), or (3) Cl's used trait is inconsistent.

Def: A specification is *consistent* if and only if it is not inconsistent.

Notice we define inconsistency of a procedure specification in terms of Th(Pr) and not Th(Pr+) so as not to include the theory of the defined type when Pr is a bound procedure specification. Since the theory of a cluster specification is defined in terms of the theories of its procedure specifications, we avoid a circular definition.

Consistency is a desirable property of all specifications. Inconsistent specifications are more common than one might imagine, as the following example illustrates.

```
intersect = proc (s1, s2: set) returns (s3: set)
   uses SetOfInt
      pre true
      post ∀j:E [has(s3↓,j) = has(s1↑,j) ∧ has(s2↑,j)]
      end
```

Intersect specifies that the set returned contains elements common to both input sets. Suppose intersect is a free procedure specification. Th(intersect) is inconsistent with respect to *SetClusSpec* (Figure 2) because there is no set object that can be returned as the intersection of disjoint input arguments. The proof of inconsistency uses *SizeInv*, derivable from the type induction rule for sets.

3.2. Full-Coverage

In this subsection and the next two, we will define three properties that are related to the "completeness" property of a specification. These three represent examples of the kinds of approximations to completeness a specifier might want to check of a specification.

A common error in programming is forgetting to cover all the cases. As a result, a program may behave in an erroneous or surprising manner on some inputs. We would like to be able to prevent the occurrence of these errors before coding begins, i.e., in the design phase, by making sure our specification covers all the cases that can arise. For example, the following specification,

```
search = proc (a: array, e: elem) returns (index: int)
   uses ArrayOfElem
      pre isSorted(a↑)
      post e↑ = fetch(a↑, index↓)
      end
```

is not fully-covering because the case for the unsorted array is not covered. A checker for full-coverage invoked on search might prompt us to add another pre/post pair to handle the unsorted array.

Unlike consistency, however, full-coverage is not always desired. We may intentionally want to leave some cases unspecified because we know they will never arise or because we want to let the programmer decide how to handle them. In the example above, we may decide not to add another pre/post pair if we expect search always to be invoked with a sorted array.

3.2.1. Definition

We want the definition of full-coverage to capture the notion that the behavior of a procedure is specified for all "reachable" input states. One way of capturing the notion of full-coverage of a procedure specification in terms of theories is that if the pre-condition of the procedure specification is equivalent to true, then the behavior is defined for all input states, and so the procedure specification is fully-covering. That is,

Def: A procedure specification, Pr, is *fully-covering* if and only if true{Pr}Pr.post is in Th(Pr+).

Def: A cluster specification is *fully-covering* if and only if all its procedure specifications are fully-covering.

3.2.2. Discussion

A specification may not appear to be fully-covering when it is. Consider *SetCluSpec* in which each of its procedure specifications, in particular, delete, is fully-covering. Although the disjunction of delete's pre-conditions is not identically true, it is provably true from Th(set), which is contained in Th(delete+). The proof would use *SizeInv* since we need not handle the case for sets of size less than or equal to zero.

In practice, writing a procedure specification that is fully-covering is similar to generating sufficient test cases for a program [Goodenough 75, McMullin 82]. A helpful guideline to follow is for the specifier to use in a stylized manner, multiple pre/post pairs in conjunction with **signals** assertions (for multiple termination conditions) to cover all the cases. If one pre-condition places a restriction on the input state, then other pre-conditions should cover the cases for which the restriction does not hold. For each separate case, there is typically a different termination condition. As a result, the behavior of the procedure is "fully" specified.

3.3. Determinism

In specifying a program, it is not always easy to separate decisions that should be made at design time from those that should be delayed to implementation time. A specification should impose as few constraints as possible to avoid unnecessarily overspecifying the behavior of the program. An intentional lack of constraint can be regarded as an intentional incompleteness.

Nondeterminism gets at the notion of introducing an intentional incompleteness in a specification. It says that the values of input and output objects of a procedure specification are not predictable in the final state. A nondeterministic specification allows the implementor the freedom to

choose the most convenient (e.g., efficient to implement) values. For example, in implementing a *choose* procedure for sets, returning the last integer inserted may be more efficient than returning the largest integer.

In contrast, *determinism* requires that the final values of the input and output objects be predictable. Whereas the fully-covering property deals with the "completeness" of a specification with respect to input states, *determinism* deals with it with respect to output states.

3.3.1. Definition

A specification is deterministic if for each state that satisfies the pre-condition, only one set of final values for the input and output objects satisfies the post-condition. We define this property in terms of theories, analogously to the usual definition for a function. A relation, f, on A X B is a (partial) function if for all a∈A, b1,b2∈B [(<a,b1>∈f ∧ <a,b2>∈f) ⇒ (b1 = b2)]. For determinism, we require the relation between the values of input an output objects defined by a procedure specification to be a partial function.

If A is the list of input formals and B is the list of output formals for the procedure specification, Pr, we let Pr.pre(A↑) denote the pre-condition on the initial values of input objects, and Pr.post(A↑, A↓, B↓) denote the post-condition on the initial and final values of input and output objects.

Def: A procedure specification, Pr, is *deterministic* if and only if Th(Pr+) contains the following formula:

$$\forall A, A1, A2, B1, B2 \; [Pr.pre(A\uparrow) \Rightarrow [Pr.post(A\uparrow, A1\downarrow, B1\downarrow) \wedge Pr.post(A\uparrow, A2\downarrow, B2\downarrow)] \Rightarrow (A1\downarrow = A2\downarrow) \wedge (B1\downarrow = B2\downarrow)]$$

3.3.2. Discussion

A specifier may intend a specification to be deterministic or not. A procedure specification may turn out to be nondeterministic because of an unintentional oversight on the part of the specifier. The following procedure specification,

```
choose = proc (s: stack) returns (i: int)
   uses StackOfInt
      pre ~isNull(s↑)
      post i↓ = top(s↑) ∧ mutates s
      end
```

is nondeterministic–the final value of s is indeterminate because of the presence of the **mutates** assertion (the final value of s may or may not be the same as its initial value). To make choose deterministic, the specifier could add the conjunct s↓ = pop(s↑) to the post-condition, or remove the **mutates** assertion. On the other hand, the specifier may have intended to let the implementer decide whether or not to pop the stack, and therefore may have intended *choose* to be nondeterministic.

Checking for determinism requires showing that a formula is in a theory; checking for nondeterminism requires showing that a formula is not. A specifier may sometimes be able to show the latter by assuming the formula is in the theory and finding a contradiction to show otherwise. For example, if the post-condition of the choose procedure specification were isIn(s↑,i↓), then we could show *choose* is nondeterministic by assuming the following formula is in Th(choose):

$$\forall s{:}stack,\ i1,i2{:}int\ [\sim isNull(s\uparrow) \Rightarrow [isIn(s\uparrow,i1\downarrow) \wedge isIn(s\uparrow,i2\downarrow)] \Rightarrow (i1\downarrow = i2\downarrow)]$$

Let s↑ be push(push(null,5),7), i1↓ be 5, and i2↓ be 7 to derive a contradiction.

3.4. Protection

By partitioning a specification into two tiers, we can avoid at the top tier an incompleteness at the bottom tier. In particular, a procedure specification should be able to use a trait even if the trait is not sufficiently-complete [Guttag 78]. It is the procedure specification's responsibility to *protect* any of its users from the incompleteness of the trait by ensuring that the meaning of the procedure specification is independent of those incompletenesses.

Associated with a trait, Tr, is a set of *exempt* terms, E(Tr), to inform the specifier of intentional incompletenesses of Tr. We would like to ensure such incompletenesses do not show through to the interface level. For example, if the term top(null) is in the set E(StackOfInt), the following procedure specification is not protective.

```
read = proc (st: stack) returns (i: int)
  uses StackOfInt
    pre true
    post i↓ = top(st↑)
    end
```

If the initial value of st were null, then the incompleteness of the stack trait would show through to the interface level because the value of the integer returned would be denoted by the exempt term top(null).

Factoring a specification into two tiers allows us to factor our checks as well. If upon checking a trait for sufficient-completeness, we discover it is not sufficiently-complete, we may be inclined to invoke our checker for protection. For example, invoking a checker on read might cause us to change its pre-condition to be ~isNull(st↑).

3.4.1. Definition

We say that a procedure specification is *protective* if it is independent of the set of exempt terms of its used trait. For a trait, Tr, the set of exempt terms, E(Tr), includes all terms that have a subterm that is provably equal to an instantiation of an exempt term. For example, for the StackOfInt trait, E(StackOfInt) = {top(null), pop(null), size(top(null)), top(pop(push(null,e))), ...}. We now give the definition of "independent of a set of terms." Intuitively, it captures the notion of never having to deal with certain terms. We follow it with the definition of protection.

Def: Let S be a set of terms. An assertion, A, appearing in Pr is *independent of S*, if

1. No subterm of A is in S, or
2. $\exists$ B [A $\Leftrightarrow$ B] $\in$ Th(Pr), and B is independent of S.

Def: Pr is *protective* if

1. Pr.pre is independent of E(Tr), and
2. Pr.pre $\Rightarrow$ Pr.post is independent of E(Tr).

3.4.2. Discussion

Protection is a desirable property of an interface specification. The specification should not be dependent on properties of the values denoted by exempt terms, and in reasoning about it the specifier does not want to be "stuck" with terms that are exempt. If upon checking to see if a specification is protective, we find that it is not, we may be able to find the dependency in the specification and then fix the specification to remove it.

Checking may require some cleverness on the specifier's part. It may involve finding an assertion equivalent to the one being shown independent of a set of exempt terms. Checking that the pre-condition is protective is usually easy because pre-conditions are usually simple. Checking the post-condition, however, is likely to be more difficult.

In practice, writing a protective procedure specification is straightforward provided that the trait is actually strong enough to specify the desired properties. Strong enough pre-conditions are written to make sure that even if a post-condition alone is not independent of an exempt term, the assertion Pre $\Rightarrow$ Post is. Often enriching the set of functions of the used trait makes it easier to read and write pre-conditions to handle these cases. For example, we might include the function isNull in the StackOfInt trait so we can write in the pre-condition of read the assertion $\sim$isNull(st↑) instead of the equivalent assertion $\sim$(st↑ = null).

4. Further Work

Two directions in which further work can be done are in building software support and finding other ways to evaluate specifications. First, from the definitions of properties of specifications, we would like to find algorithms or heuristics from which we can build software tools that enable us to check for properties of specifications. A specifier can then use these checkers to help evaluate a specification as it develops. Existing software tools that can be used to manipulate specifications include OBJ [Goguen 79], AFFIRM [Musser 80], and Reve [Lescanne 83]; proposed tools include those for Larch [Guttag 82] and for PAISLey [Zave 82].

Second, besides checking for properties, we need to find other useful means of evaluating specifications, and besides those properties we have defined, we need to find other properties or qualities for which to evaluate a specification. For example, comparing the relative strength between two specifications [Wing 83, Wing 84] or analyzing structural dependencies between components of a specification might prove to be two other valuable ways to evaluate specifications. We also need to consider the task of evaluating a collection of specifications as well as individual specifications. As a collection of specifications grows, evaluating it becomes just as important as, and probably harder than, evaluating each of its components.

I. SetOfInt Trait

```
SetOfInt: trait
   includes Integer, Equivalence
   introduces
      empty: → SI
      add: SI, E → SI
      remove: SI, E → SI
      has: SI, E → Bool
      isEmpty: SI → Bool
      card: SI → Int
   closes SI over [empty, add]
   constrains [SI] so that for all [s:SI, e, e1: E]
      remove(empty,e) = empty
      remove(add(s,e),e1) = if eq(e,e1) then remove(s,e1) else add(remove(s,e1),e)
      has(empty,e) = false
      has(add(s,e),e1) = if eq(e,e1) then true else has(s,e1)
      isEmpty(empty) = true
      isEmpty(add(s,e)) = false
      card(empty) = 0
      card(add(s,e)) = if has(s,e) then card(s) else 1 + card(s)
```

References

[Burstall 81] R.M. Burstall and J.A. Goguen: An Informal Introduction to Specifications Using CLEAR: *The Correctness Problem in Computer Science*, Academic Press, Boyer and Moore (eds.), 1981.

[Goguen 78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner: An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types: *Current Trends in Programming Methodology* (IV): Data Structuring, Prentice-Hall, 1978.

[Goguen 79] J.A. Goguen and J.J. Tardo: An Introductionto OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications: Proceedings on Specifications of Reliable Software, Boston, Mass., April 1979, pp. 170-189.

[Goodenough 75] J.B. Goodenough and S.L. Gerhart: Toward a Theory of Test Data Selection: IEEE Transactions on Software Engineering, 1(2), June 1975, pp. 156-173.

[Guttag 77] J.V. Guttag: Abstract Data Types and the Development of Data Structures: Communications of the ACM, 20(6), June 1977, pp. 396-404.

[Guttag 78] J.V. Guttag and J.J. Horning: The Algebraic Specification of Abstract Data Types: Acta Informatica (10), 1978, pp. 27-52.

[Guttag 82] J.V. Guttag, J.J. Horning, and J.M. Wing: Some Notes on Putting Formal Specifications to Productive Use: Science of Computer Programming, 2(1), October 1982, pp. 53-68.

[Guttag 83] J.V. Guttag and J.J. Horning: An Introduction to the Larch Shared Language: Proceedings of the IFIP 9th World Computer Congress, Paris, France, September

1983, pp. 809-814.

[Lescanne 83] P. Lescanne: Computer Experiments with the REVE Term Rewriting System Generator: Proceedings of the Tenth ACM Symposium on Principles of Programming Languages, Austin, Texas, January 1983, pp. 99-108.

[Liskov 81] B.H. Liskov et. al: *CLU Reference Manual*, Lecture Notes in Computer Science (114), Springer-Verlag, 1981.

[McMullin 82] P.R. McMullin: DAISTS: A System for Using Specifications to Test Implementations: Ph.D. thesis, University of Maryland, 1982.

[Musser 80] D.R. Musser: Abstract Data Type Specification in the Affirm System: IEEE Transactions on Software Engineering, 6(1), January 1980, pp. 24-32.

[Nakajima 80] R. Nakajima, M. Honda, and H. Nakahara: Hierarchical Program Specification and Verification—A Many-Sorted Logical Approach: Acta Informatica (14), 1980, pp. 135-155.

[Wing 83] J.M. Wing: A Two-Tiered Approach to Specifying Programs: MIT Laboratory for Computer Science, Cambridge, MA, Technical Report 299, 1983.

[Wing 84] J.M. Wing: Strength and Essentiality of Specifications: International Workshop on Models and Languages for Software Specifications and Design, Orlando, Florida, March 1984, (position paper).

[Zave 82] P. Zave: An Operational Approach to Requirements Specification for Embedded Systems: IEEE Transactions on Software Engineering, 8(3), May 1982, pp. 250-269.