Protection from the Underspecified

Gary T. Leavens
Department of Computer Science
Iowa State University, Ames, Iowa 50011
Jeannette M. Wing School of Computer Science
Carnegie Mellon University, Pittsburgh, Pennsylvania 15213

April 1996

CMU-CS-96-129

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted to the ACM SIGSOFT Symposium on Foundations of Software Engineering 1996.

Leavens's work is supported in part by NSF grant CCR-9593168. Wing's work is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. Views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of Wright Laboratory or the United States Government.

Keywords: Partial and total functions; Protective specifications; Specification languages; Underspecification; Larch.

Abstract

Underspecification is a good way to deal with partial functions in specification and reasoning. However, when underspecification is used, implementations may unintentionally be forced to depend on parts of the specification that were supposed to be underspecified. We show how to write pre- and postcondition specifications that avoid such problems, by having the precondition "protect" the postcondition from the effects of underspecification. This approach is most practical if the specification of mathematical vocabulary is separated from the specification of implementation behavior, as in Larch, because it gives the specifier a chance to think about protection separately from the specification of mathematical behavior. We formalize the notion of protective procedure specifications, and show how to prove that a specification is protective. We also extend the Larch Shared Language to allow specification of what is intentionally left underspecified, which also allows enhanced "debugging" of such specifications.

1 Undefinedness and Underspecification

Any method for specifying and verifying computer programs has to deal with partial functions. Our explanation of this problem reviews a recent article by C. Jones [13]. Since the use of underspecification as a solution to this problem has been advocated by others [9], our point, in this review, is the need for ways to:

- document what is intended to be "completely-defined" (or, conversely, underspecified), and
- prevent underspecification from having unintended consequences.

1.1 Background

A *partial function* is a function that does not give a value for some elements of its declared domain. For example, the operator that returns the head of a list can be modeled as a partial function on lists; if that is done, then `head(empty)` fails to denote an element. (That is, `head(empty)` is "undefined.")

One way to deal with partiality in reasoning is to use a specialized logic, for example, one with three logical values and two kinds of equality [1]. However, because all such logics either do not satisfy standard logical laws or are not compositional, such logics are subtle, and thus more difficult to use and teach [9]. More importantly, if one uses informal reasoning and informal specifications, as is common in real software projects, then there is no hope of using such a specialized logic.

We agree with Gries and Schneider [9] that the best approach to dealing with partiality is to use *underspecification*. That is, one avoids specifying a value for undefined terms, but assumes that all functions are total. As a concrete example of this approach, consider the Larch Shared Language (LSL) [10, Chapter 4], [11]. In LSL's logic, all functions are presumed to be total. That is, `head(empty)` denotes some element of the appropriate type, even if the user has not specified what element that term denotes. Where an LSL specification is silent, terms take on some (unspecified) value.

In common with other logics that use underspecification to avoid the undefined, the logic of LSL is classical, and thus has several pleasing formal properties. More importantly, classical logic matches informal reasoning.

1.2 Jones's Examples

Recently, Jones presented some "counter examples" to logics like LSL's, which use underspecification to deal with partiality [13]. We use these examples to explain LSL, underspecification, and the problem we are solving.

We translate Jones's first example into the LSL trait shown in Figure 1. This trait defines a sort, `OneElem`, a constant `it`, and a function `f`. Because of the **generated by** clause, the sort `OneElem` has only one element, the constant `it`. (The current version of LSL allows such sorts, contrary to [13].) In LSL `f(-1) = it`, because `f` has to take on some value when

```
JonesExample1: trait
  includes Integer
  introduces
    it: → OneElem
    f: Int → OneElem
  asserts
    OneElem generated by it
    ∀ i: Int
      f(i) == if i=0 then it else f(i-1)
  implies
    converts f: Int → OneElem
```

Figure 1: Jones's "counter example".

applied to `-1`, and the only possible value is `it`. Although Jones notes that this is "not an inconsistency" he says that "it is certainly likely to surprise someone who views" the definition of `f` as specifying "a partial function" (p. 66). However, the amount of surprise would be in direct proportion to the strength of the person's belief that they are defining partial functions in LSL, a view that is simply mistaken. Thus, rather than being a "counter example," we see this as an "educational example."

Although this example illustrates the technical point that all LSL functions are total, we do not expect that every user of LSL understands it. Instead, we believe that a practical specification formalism should allow the specifier to state what is intended to be "completely-defined" (or, conversely, underspecified) in such cases.

Jones's other major example brings out a more important warning about the underspecification approach. This example is a recursive definition of the factorial function, and is translated into LSL in Figure 2. Jones's warning about this example is that, in a logic such as LSL's, a model of `fact` must satisfy irrelevant equations such as the following, which is also highlighted in the redundant **implies** section of the trait.

$$\mathtt{fact}(-1) == -\mathtt{fact}(-2) \tag{1}$$

This follows because `fact(-1)` denotes some (unspecified) value. Assuming that such equations are accidents of the logic and not intended by the specifier, it would be very serious if such irrelevant properties would have to be implemented in a program.

Gries and Schneider [9], in reply to this last example, say that "the fault lies in the recursive definition rather than in handling the undefined by underspecification" (p. 373). Their specification of this example is translated into LSL in Figure 3.

However, Gries and Schneider's specification does not completely dispense with the problems that Jones's factorial example raises. The remaining problem is that users might accidentally do what Jones's factorial ex-

```
factTrait: trait
  includes Integer
  introduces
    fact: Int → Int
  asserts
    ∀ i: Int
      fact(i) == if i=0 then 1 else i * fact(i-1);
  implies
    equations
      fact(3) == 6;
      fact(-1) == - fact(-2);
```

Figure 2: Jones's factorial example.

```
factTrait2: trait
  includes Integer
  introduces
    fact: Int → Int
  asserts
    ∀ i: Int
      fact(0) == 1;
      (i > 0) ⇒ fact(i) = i * fact(i-1);
  implies
    equations
      fact(3) == 6;
```

Figure 3: Gries and Schneider's improvement to Jones's factorial example.

ample warns against: require implementations to satisfy unintended constraints. The point is that users of such a specification formalism are prone to make such mistakes, unless the formalism has some way of preventing or catching such mistakes.

Gries and Schneider's version of the specification avoids the problems Jones warns against, but they offer only education as a remedy. While we applaud their efforts, we believe that more practical solutions should not require so much sophistication, especially if they are to be used informally. Thus we advocate:

- the separation of an implementation's specification into two tiers, and
- redundant ways of specifying what is intended to be completely-defined (or underspecified).

The separation of a specification into two tiers can be used even in informal contexts [16] to guard against the problems Jones warns about. The redundant ways of specifying intent can be used to debug and check formal specifications to further guard against this problem.

In formal specifications, redundant ways of specifying intent are needed, because a logical formula may encode information about the intended domain of definition of an operator in various clever ways. Such clever encodings may make it difficult to extract the information that Gries and Schneider's definition so clearly displays. For example, each of the following formulas is equivalent to the last axiom in the **asserts** section of Figure 3 [8, Section 3.6].

```
¬(i > 0) ∨ fact(i) = i * fact(i-1);
((i > 0) ∧ fact(i) = i * fact(i-1)) = (i > 0);
¬(fact(i) = i * fact(i-1)) ⇒ ¬(i > 0);
```

1.3 Outline of the Paper

In Section 2 below we show how one can use preconditions in layered specifications, as in Larch, to protect against the consequences of underspecified mathematical vocabulary. In Section 3 we define protection and show how to prove that a behavioral interface specification is protective. In Section 4 we extend LSL to allow one to specify what is intended to be completely-defined by a trait, and then use that information to give a second proof technique for proving that a behavioral interface specification is protective. In Section 5 we offer some discussion, and we conclude with a summary in Section 6.

2 Protection from the Underspecified

The use of pre- and postconditions in a specification helps avoid imposing unintended constraints on implementations. Since one only cares about the meaning of the postcondition when the precondition is true, the precondition can "protect" the mathematical operators used in the postcondition

```
int factorial(int x) {
   requires informally "x is nonnegative and not too big";
   ensures informally "result is the factorial of x";
}
```

Figure 4: An informal version of the factorial example.

from areas of underspecification. Thus when specifying the mathematical operators, one need not be as careful as Gries and Schneider.

To make these ideas concrete, we consider the Larch family of behavioral interface specification languages (BISLs) [12, 24, 23]. In the Larch family, one specifies implementations in two tiers by describing:

- mathematical vocabulary in LSL, and
- syntactic interfaces (names, types, number of arguments, etc.), and behavior (pre- and postconditions) in a BISL.

Each BISL is tailored to specifying interface details for some specific programming language. (The interface aspects are of no concern in this paper.)

The idea of protection in a BISL was first formulated by Wing [24, Section 5.1.4]. Although we generalize that notion here, what we seek is the same as Wing's original goal: knowing when a behavioral interface specification has protected "its users from the incompleteness of the trait by ensuring that the meaning of the procedure specification is independent of" any incompleteness in the trait (p. 123).

An example of how this idea can be used informally is given in Figure 4, which specifies an integer-valued factorial procedure, which is to be implemented in C++. The informal pre- and postconditions follow **requires** and **ensures**, respectively. (The keyword **informally** in Larch/C++ [15] signals the start of an informal predicate.) In this specification the precondition requires that the argument `x` has a well-defined factorial. Since the precondition "protects" the postcondition in this manner, the (implicit) underspecification of "the factorial of `x`" when `x` is negative does not matter. (The requirement that `x` is "not too big" allows an implementation of `factorial` to guarantee termination.)

A formal version of the factorial example, also specified in Larch/C++, appears in Figure 5. The **uses** clause says that this specification is written using mathematical vocabulary drawn from (some built-in traits and) the trait `factTrait` of Figure 2. Since the suspect trait is used, does a correct implementation of Figure 5, have to satisfy irrelevant consequences of `factTrait`'s theory, such as Equation (1)? It does not, because the precondition "protects" the use of the trait's `fact` operator in the postcondition. That is, the precondition specifies that the argument must be positive. If the argument is not positive, then, as usual, the specification says nothing about the result. So if the precondition is not met, the implementation is

```
uses factTrait(int for Int);

int factorial(int x) {
   requires 0 ≤ x ∧ x ≤ 8;
   ensures result = fact(x);
}
```

Figure 5: A formal version of the factorial example.

```
imports Table, Elem, Key, NoAssociation;

Elem fetch(Table t, Key k) throw(NoAssociation) {
   ensures (defined(t, k) ⇒ result = apply(t, k))
        ∧ (¬defined(t, k) ⇒ thrown(NoAssociation) = theException);
}
```

Figure 6: Protective specification of a C++ member function that fetches an element from a table.

under no obligation to meet the postcondition, and hence need not worry about the consequences of Equation (1). More importantly, if the precondition is met, then the implementation need not worry about the consequences of Equation (1), because the argument will be nonnegative.

In programs, one often wants to raise (throw) an exception when some boundary condition is violated. One way to write such a specification is shown in Figure 6. In this figure, the abstract values of `Table` objects are taken from the sort `M` in the trait `FiniteMap` [10, p. 185]. (A part of this trait is shown in Figure 7.) Note that the operator `apply` is underspecified and that `defined` is simply another operator from the trait `FiniteMap`. The term "**thrown**`(NoAssociation) = theException`" in the postcondition means that an exception is raised, and that the exception's type is `NoAssociation`, and that the exception result's abstract value is `theException`.

In Figure 6, there are, in essence, two preconditions, `defined(t, k)` and its negation, and the first of these is protecting the application of `apply`. This postcondition is well-defined, even though `apply` is underspecified, because `apply` takes on some (arbitrary) value for the empty table and because LSL uses classical logic.

As an aside, users of a specification language might still worry that the postcondition of Figure 6 is "undefined" if part of it is. This worry can be alleviated in Larch/C++ by a syntactic sugar that allows one to split the specification up into multiple cases, each with its own precondition. This sugar would allow the specification in Figure 6 to be written as in Figure 8. Such multiple pre- and postconditions are similar to guarded

```
FiniteMap(M,D,R): trait
  introduces
    {}: → M
    update: M, D, R → M
    apply: M, D → R
    defined: M, D → Bool
  asserts
    M generated by {}, update
    M partitioned by apply, defined
    ∀ m: M,  d,d1,d2: D,  r: R
       apply(update(m,d2,r), d1) == if d1=d2 then r else apply(m,d1);
       ¬defined({}, d);
       defined(update(m,d2,r), d1) == d1=d2 ∨ defined(m,d1)
  implies
    converts apply, defined exempting ∀ d: D apply({}, d)
```

Figure 7: Part of the `FiniteMap` trait, quoted from [10, p. 185].

```
imports Table, Elem, Key;

Elem fetch(Table t, Key k) throw(NoAssociation) {
   requires defined(t, k);
   ensures result = apply(t, k);

   requires ¬defined(t, k);
   ensures thrown(NoAssociation) = theException;
}
```

Figure 8: A sugared version of the previous specification of `fetch`. An implementation has to satisfy both pre- and postcondition pairs.

commands [4], and have been used in the specification languages Larch/CLU [24, Section 4.1.4], and Fresco [20, 22, 21]. Besides bringing the issue of protection to the specifier's attention, this notational convenience makes it easier to automatically check that a specification is protective. Without such notation, the specification of Figure 6 could be expressed in various logically equivalent ways that would make it hard to see what is being protected.

We believe that, during interface specification, a specifier would normally think about such issues as whether the mathematical operators are well-defined on their arguments when writing the precondition of such procedure specifications. This belief is based on our experience in writing behavioral interface specifications, and in teaching students to write such specifications. The use of two tiers with preconditions for specifying implementations thus meshes well with the underspecification approach, because the values of mathematical operators outside their domains are not needed to understand or implement such a specification.

In the rest of this paper we explore the notion of protection, with the aim of improving intuition about protection and providing more guidance to specifiers. To this end we concentrate on formal specifications below.

3 Completely-defined and Protective Specifications

In this section we formally define the notion of protection in a BISL using the more primitive notion of a completely-defined term.

We say that an LSL term is completely-defined if it can be proved to have the same value in all models of its trait. To state this condition formally, we use a variation of an idea found in the Larch Prover for proving that an operator is "converted" [10, pp. 142–4].

Let T be a trait. Let T' be a version of the trait T with every operator f in T replaced by f', except that the following operators are left alone:

- all operators in the built-in trait `Boolean`,
- all operators in all instances of the built-in traits `Conditional` (which specifies **if then else**), and `Equality` (which specifies the operators $=$ and $\neq$), and
- all operators mentioned in a **generated by** clause.

For example, the trait `factTrait`$'$ has `fact` replaced by `fact`$'$, but `true` and the boolean operators are not primed, and neither are `0`, `pred`, and `succ`, because they are mentioned in the **generated by** clause of the trait `Integer` [10, p. 161]. (Operators mentioned in a **generated by** clause are meant to be a canonical way to describe values of a given sort; two ways to describe such values cannot both be canonical.)

Similarly, if P is a term in the language of T, then let P' be a copy of P with every operator f that appears in P replaced by f', with the same exceptions as above. For example, if P is "**result** $=$ `fact(x)`", then P' would be "**result** $=$ `fact`$'$`(x)`", because `fact` is not exempted from priming, "$=$" is exempt from priming, and **result** and `x` are not operators.

In what follows, we write $T \vdash P$ to mean that P is provable from trait T.

Definition 3.1 (completely-defined) *An LSL term $P(\vec{x})$, with free variables $\vec{x}$ of sorts $\vec{U}$, is* completely-defined *for trait T if and only if $T \cup T' \vdash \forall \vec{x} : \vec{U} \,.\, P(\vec{x}) = P'(\vec{x})$.*

Trivial examples of completely-defined terms include variables, because in each trait T, $T \vdash \forall x : U \,.\, x = x$. A more interesting example is that, for `factTrait`, the term `fact(27)` is completely-defined, but both `fact(-1)` and `fact(x)`, where `x:Int`, are not.

Just because a term is not completely-defined does not mean it is "bad". For example, the term `choose({1}` $\cup$ `{2})` is not completely-defined for the trait `ChoiceSet` (of [10, p. 176]).

The following definition of when a procedure specification is protective says, in essence, that the precondition must be completely-defined for the used trait, and that whenever the precondition holds, then the postcondition must be completely-defined for the used trait.

Definition 3.2 (protective) *A procedure specification, S, that uses trait T is* protective *if for each pair of precondition $Q(\vec{x})$ and postcondition $R(\vec{x})$,*

- $T \cup T' \vdash \forall \vec{x} : \vec{U} \,.\, Q(\vec{x}) = Q'(\vec{x})$, *and*
- $T \cup T' \vdash \forall \vec{x} : \vec{U} \,.\, Q(\vec{x}) \Rightarrow (R(\vec{x}) = R'(\vec{x}))$.

The definition of a protective procedure specification suggests a direct proof technique. For example, to prove that the specification of `factorial` in Figure 5 is protective, one must show that $\texttt{factTrait} \cup \texttt{factTrait}'$ proves both of the following:

- $\forall \texttt{x} : \texttt{int} \,.\, (0 \leq \texttt{x} \wedge \texttt{x} \leq 8) = (0 \leq' \texttt{x} \wedge \texttt{x} \leq' 8')$, and
- $\forall \texttt{x} : \texttt{int}.(0 \leq \texttt{x} \wedge \texttt{x} \leq 8) \Rightarrow (\textbf{result} = \texttt{fact}(\texttt{x})) = (\textbf{result} = \texttt{fact}'(\texttt{x}))$.

Although such proofs are straightforward, they are quite tedious.

4 Proving Protection in a BISL

In this section we describe an easier way to prove protection in a BISL. This proof technique uses extra information that specifiers would add to LSL traits. This extra information would also allow a user of LSL to more precisely specify and check what is intended to be completely-defined.

4.1 Specifying What is Not Underspecified

LSL already has some provision for specifying what is not underspecified — the specification of when an operator is "converted". This is done by using a **converts** clause, as was done in Figure 1. A **converts** clause says that the axioms of the trait uniquely define the operators named in the clause, "relative to the other operators in the trait" [10, p. 142]. (See the appendix for a more complete explanation of conversion.)

Unfortunately, proving that an LSL operator is converted does not mean it is completely-defined; it may still be underspecified. For example, consider the trait in Figure 9. In this trait, the operator `somewhatBigger` is defined to be equal to `muchBigger`; however, `muchBigger` is quite underspecified, since no assertions constrain it. Yet, the **converts** clause in the **implies** section is still provable, because `somewhatBigger` is completely-defined, relative to `muchBigger`. That is, once `muchBigger` is determined, `somewhatBigger` becomes completely-defined.

Because of this distinction between conversion and complete definition we propose adding another implication clause to LSL. This clause, which we call the **exact** clause, has a form similar to that of the LSL **exempting** clause (although it would not be a subclause of a **converts** clause). The

```
biggerTrait: trait
  includes Integer
  introduces
    muchBigger, somewhatBigger: Int → Int
  asserts
    ∀ i: Int
       somewhatBigger(i) == muchBigger(i);
  implies
    converts somewhatBigger: Int → Int
```

Figure 9: An LSL trait in which `somewhatBigger` is convertible, but `somewhatBigger(i)` is not completely-defined.

```
factTrait: trait
  includes Integer
  introduces
    fact: Int → Int
  asserts
    ∀ i: Int
      fact(i) == if i=0 then 1 else i * fact(i-1);
  implies
    equations
      fact(3) == 6;
    exact ∀ k: Int such that k ≥ 0
      fact(k)
```

Figure 10: Factorial example demonstrating the **exact** clause.

idea is that it would allow one to specify terms that should be completely-defined. For example the **exact** clause in Figure 10 says that terms of the form `fact(k)` are intended to be completely-defined, if `k` $\geq$ `0`.

The extra information in the **exact** clause can be used to help debug an LSL specification, by trying to prove the following property.

Definition 4.1 (exact for T) *Let T be a trait that contains an* **exact** *clause of the form* **exact** $\forall \vec{a} : \vec{A}$ **such that** $Q(\vec{a})\ P(\vec{a})$*, where $Q(\vec{a})$ is a predicate and $P(\vec{a})$ is a term in the language of T. This clause is* exact for T *if and only if:*

$$T \cup T' \vdash \forall \vec{a} : \vec{A} \,.\, (Q(\vec{a}) \wedge Q'(\vec{a})) \Rightarrow P(\vec{a}) = P'(\vec{a}). \qquad (2)$$

For example, in Figure 10, the **exact** clause is exact for `factTrait` if the following condition is provable from $\mathtt{factTrait} \cup \mathtt{factTrait}'$.

$$\forall \mathtt{k} : \mathtt{Int} \,.\, (\mathtt{k} \geq 0 \wedge \mathtt{k} \geq' 0) \Rightarrow \mathtt{fact}(\mathtt{k}) = \mathtt{fact}'(\mathtt{k}).$$

The proof would proceed by induction on `k`.

$\texttt{Exact('}x\texttt{')} = \texttt{true}$, if x is a variable

$\texttt{Exact('}P(\vec{E})\texttt{')} = \bigwedge_{E_i \in \vec{E}} \texttt{Exact('}E_i\texttt{')} \wedge Q(\vec{E})$,
if the trait's **implies** section contains a clause:
exact $\forall \vec{a} : \vec{A}$ **such that** $Q(\vec{a})$ $P(\vec{a})$

$\texttt{Exact('}\neg E\texttt{')} = \texttt{Exact('}E\texttt{')}$

$\texttt{Exact('}E_1 \circ E_2\texttt{')} = \texttt{Exact('}E_1\texttt{')} \wedge \texttt{Exact('}E_2\texttt{')}$,
if $\circ$ is $=$, $\neq$, or a boolean operator: $\wedge$, $\vee$, or $\Rightarrow$

$\texttt{Exact('}\forall \vec{x} : \vec{T} \,.\, E\texttt{')} = \forall \vec{x} : \vec{T} \,.\, \texttt{Exact('}E\texttt{')}$

$\texttt{Exact('}\exists \vec{x} : \vec{T} \,.\, E\texttt{')} = \forall \vec{x} : \vec{T} \,.\, \texttt{Exact('}E\texttt{')}$

$\texttt{Exact('}$**if** E_1 **then** E_2 **else** $E_3\texttt{')} = \texttt{Exact('}E_1\texttt{')}$
$\wedge\ \texttt{Exact('}E_2\texttt{')} \wedge \texttt{Exact('}E_3\texttt{')}$

$\texttt{Exact('}E\texttt{')} = \texttt{false}$, otherwise

Figure 11: Definition of `Exact`.

4.2 Exact Predicates

For use in proving protection, we define predicates of the form $\texttt{Exact('}E\texttt{')}$, based on the form (text) of each expression E. (These resemble the domain predicates, $\texttt{Dom('}E\texttt{')}$, described by some authors [7, 3, 2]. However, they have a different purpose, since an operator, such as `choose` on nonempty sets, may be underspecified for a reason other than being partial.) The definition of `Exact('·')` is based on the **exact** clauses given in the trait's implications (and those of included traits). This definition is lifted to arbitrary terms by requiring terms substituted for the variables in an **exact** clause to be themselves exact, and using the structure of terms formed from LSL's built-in trait operators (boolean operators, equality, and conditionals). See Figure 11 for the definition.[1]

For example, for the trait of Figure 10, the following holds.

$$\texttt{Exact('fact(i)')} = (\texttt{i} \geq \texttt{0})$$

As another example, as no **exact** clause is given in Figure 1, `Exact('f(i)')` is `false`, even though the term is completely-defined.

4.3 Using Exact Predicates to Prove Protection

Provided the information given in the **exact** clauses are exact for a trait T, then `Exact` predicates can be used as a sufficient condition for determining when a term is completely-defined for T.

Lemma 4.2 *Let T be a trait in which each* **exact** *clause is exact for T. Let $R(\vec{x})$ be a term with free variables $\vec{x} : \vec{U}$. If $T \vdash \forall \vec{x} : \vec{U} \,.\, \texttt{Exact('}R(\vec{x})\texttt{')}$, then $R(\vec{x})$ is completely-defined for T.*

[1] The free variables of these terms are not important, so they are suppressed.

Proof: (by induction on the structure of terms). Let $R(\vec{x})$ be such that $T \vdash \forall \vec{x} : \vec{U} \,.\, \texttt{Exact('}R(\vec{x})\texttt{')}$.

For the basis, suppose $R(\vec{x})$ is a variable x_i. Then $\forall \vec{x} : \vec{U} \,.\, x_i = x_i$ is trivially provable, and so x_i is completely-defined by definition.

For the inductive step, suppose that the result holds for all subterms of $R(\vec{x})$. If $R(\vec{x})$ is an invocation of some operator of T that is not a boolean operator, equality, inequality, or **if then else**, then by definition, it must be that $R(\vec{x})$ has the form $P(\vec{E}(\vec{x}))$ and that trait T has a clause of the form **exact** $\forall \vec{a} : \vec{A}$ **such that** $Q(\vec{a})$ $P(\vec{a})$. Furthermore, by definition of `Exact('·')`, it must be the case that

$$T \vdash \bigwedge_{E_i(\vec{x}) \in \vec{E}(\vec{x})} \texttt{Exact('}E_i(\vec{x})\texttt{')} \wedge Q(\vec{E}(\vec{x})). \tag{3}$$

Since T' is a primed copy of T, it must also be the case that

$$T' \vdash \bigwedge_{E'_i(\vec{x}) \in \vec{E}'(\vec{x})} \texttt{Exact('}E'_i(\vec{x})\texttt{')} \wedge Q'(\vec{E}'(\vec{x})). \tag{4}$$

Because the $\vec{x}$ are free in the above two formulas, by universal generalization

$$T \cup T' \vdash \forall \vec{x} : \vec{U} \,.\, Q(\vec{E}(\vec{x})) \wedge Q'(\vec{E}'(\vec{x})). \tag{5}$$

By the inductive hypothesis, since each $E_i(\vec{x})$ is exact, for each i,

$$T \cup T' \vdash \forall \vec{x} : \vec{U} \,.\, E_i(\vec{x}) = E'_i(\vec{x}). \tag{6}$$

Since the **exact** clauses are assumed to be exact for T, by definition we have

$$T \cup T' \vdash \forall \vec{a} : \vec{A} \,.\, (Q(\vec{a}) \wedge Q'(\vec{a})) \Rightarrow P(\vec{a}) = P'(\vec{a}). \tag{7}$$

Instantiating $\vec{a}$ to $\vec{E}(\vec{x})$, and using Formula (6), it follows that

$$T \cup T' \vdash \forall \vec{x} : \vec{U} \,.\, (Q(\vec{E}(\vec{x})) \wedge Q'(\vec{E}'(\vec{x}))) \Rightarrow P(\vec{E}(\vec{x})) = P'(\vec{E}'(\vec{x})) \tag{8}$$

But by (5), the hypothesis of this implication is provable, so $T \cup T' \vdash \forall \vec{x} : \vec{U} \,.\, P(\vec{E}(\vec{x})) = P'(\vec{E}'(\vec{x}))$ follows.

The other cases follow directly from the inductive hypothesis and the definition of `Exact('·')`. ∎

However, the converse to the above lemma does not hold. One reason is that the specifier of the used trait may not note when some terms are exact. But even if the information given is complete, the definition of `Exact` does not take into account other knowledge from the theory of the trait. For example, consider the trait `bufferTrait`, which is specified in Figure 12. It specifies the constant `bufSize`, but `bufSize` is underspecified (hence no **exact** clause is given). The term

```
bufSize < 4096
```

is completely-defined for `bufferTrait`. However,

```
Exact('bufSize < 4096') = false,
```

```
bufferTrait: trait
  includes Integer
  introduces
    bufSize: → Int
  asserts
    equations
      0 < bufSize ∧ bufSize ≤ 1024;
```

Figure 12: A trait with an underspecified constant.

because `Exact('bufSize')` is `false`.

Definition 4.3 (exact procedure specification) *A procedure specification, S, that uses trait T is* exact *if for each pair of precondition $Q(\vec{x})$ and postcondition $R(\vec{x})$,*

- $T \vdash \forall \vec{x} : \vec{U} \,.\, \texttt{Exact('}Q(\vec{x})\texttt{')}$*, and*
- $T \vdash \forall \vec{x} : \vec{U} \,.\, Q(\vec{x}) \Rightarrow \texttt{Exact('}R(\vec{x})\texttt{')}$.

Our suggested technique for proving that a procedure specification is protective, therefore, is to prove that it is exact.

Corollary 4.4 *Let T be a trait in which each* **exact** *clause is exact for T. Let S be a procedure specification that uses trait T. If S is exact, then S is protective.*

Proof: Let $Q(\vec{x})$ be the precondition of S, and let $R(\vec{x})$ be its postcondition. Suppose S is exact. Then by definition, $T \vdash \forall \vec{x} : \vec{U} \,.\, \texttt{Exact('}Q(\vec{x})\texttt{')}$. So by Lemma 4.2, $Q(\vec{x})$ is completely-defined for T. Also by definition, $T \vdash \forall \vec{x} : \vec{U} \,.\, Q(\vec{x}) \Rightarrow \texttt{Exact('}R(\vec{x})\texttt{')}$. Suppose for each $\vec{x}$, $Q(\vec{x})$ holds. Then, for each $\vec{x}$, $\texttt{Exact('}R(\vec{x})\texttt{')}$ holds, and so by Lemma 4.2, $R(\vec{x})$ is completely-defined for T. ∎

As an example of the use of the above corollary, we show how to prove that the specification of `factorial` in Figure 5 is completely-defined with respect to the trait in Figure 10. To do this we prove that the specification is exact with respect to the trait in Figure 10. First, the precondition is exact, because `Exact('x ≥ 0')` is `true`. (`Exact('0')` is `true`, because `0` is a generator. We assume the trait `Integer` has been extended with implications that say that $\geq$ is exact.) Then for the postcondition, one can calculate as follows, for all `x : int`.

```
  x ≥ 0 ⇒ Exact('result = fact(x)')
= {by definition of Exact}
  x ≥ 0 ⇒ (Exact('result') ∧ Exact('fact(x)')
= {by definition of Exact for fact, treating result as a variable}
  x ≥ 0 ⇒ (true ∧ true ∧ true ∧ x ≥ 0)
= {by predicate calculus}
  true
```

```
void chaos1(int& x) {
   modifies x;
   ensures true;
}
```

Figure 13: The Larch/C++ specification of a procedure that is protective, even exact, but not deterministic.

```
uses bufferTrait;
int foo(int x) {
   requires bufSize < x;
   ensures result = 3;
}
```

Figure 14: A specification that is deterministic but not protective.

However, if a procedure specification is protective, it is not necessarily exact. For example, a specification that uses the term `bufSize < 4096` as its precondition could be protective without being exact. Thus exactness is a sufficient, but not necessary, condition for protection.

As another example, consider the specifications of the `fetch` procedure in Figures 6 and 8. Suppose the following **exact** clause were added to the trait `FiniteMap` in Figure 7.

```
exact ∀ m:M, d:D such that defined(m, d)
  apply(m, d)
```

Then the specification of Figure 6 is protective, but not exact for this augmented `FiniteMap` trait. However, the specification of Figure 8 is exact.

5 Discussion

One might wonder whether a procedure specification is protective if and only if it is deterministic. However, the two notions are orthogonal. For example, the specification given in Figure 13 is protective (even exact) but very nondeterministic. It specifies a C++ procedure that can change the value of the object `x` (passed by reference) to any integer. Figure 14 is an example of a specification that is not protective, because the precondition is not completely-defined, but the procedure specified must be deterministic when its precondition is met.

One might think that, if one could avoid incompleteness at the trait level, then one would not need the underspecification approach at all. The problem is how to complete an LSL specification in a general way (i.e., one for which some sensible logic exists). One might imagine using an initial (or final) semantics for LSL traits, since such a semantics is a general way to complete equational specifications. However, such initial (or final) algebras do not exist, in general, for specifications in LSL [25, Section 5.4], because LSL traits can contain **generated by** clauses, which act as hierarchy conditions. (For example, an initial algebra for `factTrait` in Figure 2, would contain nonstandard integers, such as `fact(-1)`, which are prohibited by the **generated by** clause in the LSL `Integer` trait.) Although for LSL, and other specification languages with powerful hierarchy constructs, such a completion of the specification is impossible in general, some specification languages do have general ways to complete specifications (e.g., VDM-SL [14] and COLD [5]). Such completions usually take advantage of the fact that besides proper values, each type in a programming language can be thought of as having at least one improper value ($\perp$), which is used to model computations of that type that go into infinite loops or cause errors. However, to avoid overspecification (i.e., not allowing a specified procedure to terminate normally with a proper value) such languages tend to have either a complex semantics for procedure specifications or a specialized logic.

One might think that in a specification language such as VDM-SL [14] or COLD [5], in which the specification logic does not use the underspecification approach, the concept of protection is not useful. But even in such a language, one could use the precondition to protect the postcondition from non-classical or non-compositional features of the logic, or from the more subtle notions in the logic. Since such specifications would be less dependent on the logical details, we believe that they would be clearer.

In PVS [18], the logic uses total functions, but each function has a domain that is precisely defined using a predicate. Our use of **exact** clauses is similar, but allows one to say what terms are intended to be completely-defined, not just what the domain of an operator is. In PVS an attempt to apply a mathematical operator outside its domain would be a type error. If PVS were used as the mathematical basis for a BISL, then one would be forced to write protective specifications in order to prevent type errors in post-conditions.

For Z [19], it seems that the draft standard has adopted the underspecification approach [26]. To apply our ideas to Z, then, one would define BISLs that use Z instead of LSL as the mathematical tier, and then the notion of protection would be used in the BISLs.

Protection is also a useful concept when coupled with executable specifications. In a language like Eiffel [17], having a precondition be flagged as false helps debugging more than having an error occur in the body of a procedure or an executable postcondition.

6 Summary

In this paper we have shown that the Larch approach to behavioral interface specification has significant advantages in avoiding potential problems caused by underspecification. Using separate tiers for the specification of

Level	Facts	
Trait	exact $\Rightarrow$ completely-defined	Lemma 4.2
	completely-defined $\not\Rightarrow$ convertible	Figure 9
BISL	exact $\Rightarrow$ protective	Corollary 4.4
	protective $\not\Rightarrow$ deterministic	Figures 13 and 14

Table 1: Summary of concepts discussed in this paper.

mathematical operators and procedure implementations allows procedure specifications to protect implementations from dependence on underspecified mathematical operators. Thus the Larch approach mitigates the problems Jones warned about [13].

Our technical results are summarized in Table 1. The main concept is when a BISL procedure specification is protective, in the sense that it does not force implementations to satisfy unintended consequences of an LSL trait. We have given two proof techniques for proving protection, one of which is equivalent to the definition (based on the notion of completely-defined terms), and a sufficient (but not necessary) test based on the notion of exact terms that is easier to apply. The concept of an exact term is based on an extension to LSL that allows one to specify which terms are not intended to be underspecified. This extension to LSL provides better documentation and allows enhanced debugging (in the sense of [6] [10, Chapter 7]) of LSL specifications.

Although, for concreteness, these ideas have been presented in the context of Larch, they could be adapted to other formal specification languages that use underspecification.

Acknowledgments

Thanks to Adrian Fiech, Clyde Ruby, Krishna Kishore Dhara, and Matt Markland for comments and discussions of earlier drafts. Thanks to Daniel Jackson for discussing partial functions in Z with me (Wing) and to him and Jim Woodcock for letting me peek at their e-mail exchange on that topic. This manuscript is submitted for publication with the understanding that the U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notation thereon.

References

[1] H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21(3):251–269, October 1984.

[2] Andrzej Blikle. The clean termination of iterative programs. *Acta Informatica*, 16:199–217, 1981.

[3] D. Coleman and J. W. Hughes. The clean termination of Pascal programs. *Acta Informatica*, 11:195–210, 1979.

[4] E. W. Dijkstra. Guarded commands, nondeterminancy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[5] L. M. G. Feijs and H. B. M. Jonkers. *Formal Specification and Design*, volume 35 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1992.

[6] Stephen J. Garland, John V. Guttag, and James J. Horning. Debugging Larch Shared Language specifications. *IEEE Transactions on Software Engineering*, 16(6):1044–1057, September 1990.

[7] Steven M. German. Automating proofs of the absence of common runtime errors. In *Conference record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 105–118. ACM, January 1978.

[8] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer-Verlag, New York, N.Y., 1994.

[9] David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, number 1000 in Lecture Notes in Computer Science, pages 366–373. Springer-Verlag, New York, N.Y., 1995.

[10] 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, N.Y., 1993.

[11] John V. Guttag, James J. Horning, and Andrés Modet. Report on the Larch Shared Language: Version 2.3. Technical Report 58, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, April 1990. Order from src-report@src.dec.com.

[12] John V. Guttag, James J. Horning, and Jeannette M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5), September 1985.

[13] C.B. Jones. Partial functions and logics: A warning. *Information Processing Letters*, 54(2):65–67, 1995.

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

[15] Gary T. Leavens. Larch/C++ Reference Manual. Version 4.1. In ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz or on the world wide web at the URL http://www.cs.iastate.edu/~leavens/larchc++.html, December 1995.

[16] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* The MIT Press, Cambridge, Mass., 1986.

[17] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, N.Y., 1988.

[18] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[19] J. M. Spivey. *Understanding Z: a Specification Language and its Formal Semantics.* Cambridge University Press, New York, N.Y., 1988.

[20] Alan Wills. Capsules and types in Fresco: Program validation in Smalltalk. In P. America, editor, *ECOOP '91: European Conference on Object Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 59–76. Springer-Verlag, New York, N.Y., 1991.

[21] Alan Wills. Refinement in Fresco. In Kevin Lano and Howard Houghton, editors, *Object-Oriented Specification Case Studies*, chapter 9, pages 184–201. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[22] Alan Wills. Specification in Fresco. In Susan Stepney, Rosalind Barden, and David Cooper, editors, *Object Orientation in Z*, Workshops in Computing, chapter 11, pages 127–135. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.

[23] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[24] Jeannette Marie Wing. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

[25] Martin Wirsing. Algebraic specification. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 13, pages 675–788. The MIT Press, New York, N.Y., 1990.

[26] Jim Woodcock and Daniel Jackson. About the semantics of partial functions in Z. Personal communication, April 1996.

A Appendix: Conversion and an Extension to LSL

This appendix explains the notion of conversion in LSL, and also presents an extension to LSL that makes the specification of conversion more expressive.

A.1 Conversion

In a LSL trait, one can state redundant properties (theorems) that one believes do (or should) hold. These redundant properties are stated in the **implies** section of the specification. Proofs of such properties can be attempted, and are a way of debugging the trait [6] [10, Chapter 7].

For our purposes, the most interesting kind of redundant property one can state in the **implies** section is that an operator is well-defined with respect to other operators. This is done by using a **converts** clause, as was done in Figure 1. A **converts** clause says that the axioms of the trait uniquely define the operators named in the clause, "relative to the other operators in the trait" [10, p. 142]. To prove this, one must show it for all possible arguments. The Larch Prover (LP) uses the following proof technique [10, pp. 142–4]. Let $T(\vec{f})$ be a trait, which names operators $\vec{f}$ in **converts** clauses in its **implies** section. Let $T(\vec{f'})$ be a version of the trait $T(\vec{f})$ in which each of the operators f_i named in a **converts** clause is replaced by f'_i. Then one proves, for each such $f_i : \vec{A} \to B$,

$$Th(T(\vec{f}) \cup T(\vec{f'})) \vdash \forall \vec{a} : \vec{A} \,.\, f_i(\vec{a}) = f'_i(\vec{a}). \tag{9}$$

The proof would show that there cannot be two different interpretations of the operator f_i.

For example, to prove the **converts** clause for `f` in Figure 1, one axiomatizes an operator `f'` in the same way as `f`, and then proves the following.

```
∀ i: Int  f(i) == f'(i)
```

(This is proved by using the rule given by the **generated by** clause in Figure 1.)

Often one wants to prove that an operator is converted, except for some arguments. For example, one would want to prove that the `head` operator on lists is converted, except that `head(empty)`, which is purposely left underspecified. To do this one uses a **converts** clause of the following form in LSL.

```
converts
   head: List[T] → T
     exempting head(empty)
```

The **exempting** clause allows the specifier to state what terms are intentionally underspecified. In terms of the proof that `head` is converted, except where it is not intentionally underspecified, the exempting clause allows one to use the following equation

```
head(empty) == head'(empty)
```

in the proof that, for all lists `l`, `head(l) == head'(l)`.

```
factTrait: trait
  includes Integer
  introduces
    fact: Int → Int
  asserts
    ∀ i: Int
      fact(i) == if i=0 then 1 else i * fact(i-1);
  implies
    ∀ i: Int
      fact(3) == 6;
    converts
      fact: Int → Int
       exempting ∀ k: Int such that k < 0
         fact(k)
```

Figure 15: A trait demonstrating the extended **exempting** clause.

A.2 An extension to LSL

The **exempting** clause in the current LSL [10, Chapter 4] [11] does not have enough expressive power to state, in general, what is left underspecified. One can only exempt a class of terms that are described by constants or universally quantified variables. For example, one cannot specify that `fact` in Figure 2 is intentionally underspecified by adding an **exempting** clause, because the current LSL only allows one to specify that constants, or all integers, are exempted. That is, there is no way to say that only the negative integers are exempted.

We propose extending LSL by allowing domain predicates for the variable declarations in an **exempting** clause. For example, we would allow the **exempting** clause of the trait given in Figure 15. This form of the **exempting** clause allows one to specify the intended exemptions with an arbitrary (boolean-valued) LSL term.[2]

The extension to the LP proof technique for proving the **converts** clause in Figure 15 is simple. The **exempting** clause gives one the following formula

```
∀ k: Int
   (k < 0) ⇒ fact(k) == fact'(k)
```

which one can use in the proof that, for all integers `i`, `fact(i) == fact'(i)`. Given that `fact'` is axiomatized with a copy of the axioms for `fact`, this allows one to prove that `fact` is converted where it is not intentionally underspecified.

This extension to LSL increases its expressive power by its ability to state redundant and checkable information.

[2] There is logical problem if the predicate following **such that** uses an operator being specified as converted. The simplest thing to do is not to allow the use of such operators in the domain predicate (following **such that**).