Protective Interface Specifications

Gary T. Leavens and Jeannette M. Wing

TR #96-04c
April 1996, Revised October, December 1996, February 1997

Keywords: Protective specifications; Specification languages; Underspecification; Partiality; Larch.

1996 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — languages, theory, Larch, VDM, Z, RESOLVE D.2.7 [*Software Engineering*] Distribution and Maintenance — documentation; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — assertions, logics of programs, pre- and post-conditions, specification techniques, theory, LSL

A version of this paper, without the appendix sections, will appear in the proceedings of the *Colloquium on Formal Approaches in Software Engineering (FASE), TAPSOFT '97, Lille, France, April 1996*, to be published by Springer-Verlag in their *Lecture Notes in Computer Science* Series.

An earlier version of this paper was called "Protection from the Underspecified". This technical report is also CMU-CS-96-129R.

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

Protective Interface Specifications

Gary T. Leavens*[1] and Jeannette M. Wing†[2]

[1] Department of Computer Science, Iowa State University, Ames, IA 50011 USA
[2]Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213 USA

Abstract

The interface specification of a procedure describes the procedure's behavior using pre- and postconditions. These pre- and postconditions are written using various functions. If some of these functions are partial, or underspecified, then the procedure specification may not be well-defined.

We show how to write pre- and postcondition specifications that avoid such problems, by having the precondition "protect" the postcondition from the effects of partiality and underspecification. We formalize the notion of protection from partiality in the context of specification languages like VDM-SL and COLD-K. We also formalize the notion of protection from underspecification for the Larch family of specification languages, and for Larch show how one can prove that a procedure specification is protected from the effects of underspecification.

1 The Problem

This paper seeks to explain and precisely define properties of "good" procedure specifications. These properties say when the precondition of a procedure specification protects the postcondition from partiality or underspecification in the vocabulary used in the specification. While we will precisely define protection for formal specifications, it can be applied and used in even informal specifications (with, of course, less precision).

To explain what a protective specification is, we start with an informal example. Consider an (ill-defined) specification of an integer-valued factorial procedure, such as that found in Figure 1. This behavioral interface specification is to be implemented in C++, which explains the C++ syntax used to specify how it is to be called. The pre- and postconditions follow **requires** and **ensures**, respectively; when the precondition is satisfied, the procedure must terminate in a state that satisfies

*Leavens's work was supported in part by NSF grant CCR-9593168, a faculty improvement leave from Iowa State, and the Computer Science and Engineering Department at the University of Washington.

†Wing's research 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.

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

Figure 1: An ill-defined informal specification of a factorial procedure.

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

Figure 2: A protective informal specification of a factorial procedure.

the postcondition. (The keyword **informally** in Larch/C++ [21] signals the start of an informal predicate.) This specification is ill-defined, because it is not clear what the procedure should return when `x` is negative. The problem is that mathematics does not define what "the factorial of `x`" means when `x` is negative, but for that case the specification seems to require a correct implementation to return some integer. Note that the problem with this specification has nothing at all to do with the particular mathematical formalism used to write the pre- and postconditions, or with any particular logic for reasoning about what they mean.

A better, yet still informal, specification of the factorial procedure is given in Figure 2. In this specification the precondition requires that the argument `x` is nonnegative, and thus has a well-defined factorial. We say that the precondition of Figure 2 "protects" the postcondition, because for all values of the arguments that satisfy the precondition, the vocabulary used in the post-condition is well-defined. Thus whatever the phrase "the factorial of x" might mean when `x` is negative does not matter.

The concept of protection, even in informal specifications, does have one subtle twist. It is that one part of a precondition may protect other parts of the precondition itself, so that the entire precondition is well-defined. Most programmers are familiar with examples where they must check that a number is nonzero before using checking some condition involving a ratio or modulo calculation. The same idea applies in specifications such as the one in Figure 3, where the first conjunct in the precondition ("denom is positive") protects the second. That is, if the first conjunct is false, the entire precondition is false, and so the meaning of the second conjunct does not matter, as the implementation will not have any specified behavior in such a case. (Note that the postcondition is also protected by the first conjunct in the precondition.)

In the example of Figure 3, the (informal) logic used to reason about the meaning

```
double taxFor(int base, int num, int denom);
behavior {
  requires informally "denom is positive and 0 ≤ (num/denom) ≤ 1";
  ensures informally "result is approximately (num/denom) * base";
}
```

Figure 3: A protective specification that demonstrates protection within the precondition.

of the precondition matters. In our informal argument we assumed that if the first conjunct in the precondition is false, then the entire precondition is false (and hence well-defined). However, since the precondition is informal, one could plausibly argue that since the "/" operator used in the second conjunct is partial, it has no meaning when "denom" is zero, and in that case perhaps the entire precondition should be considered meaningless. To resolve such questions, one must take the first step towards a formal specification language, and agree on some conventions for interpreting such formulas.

In this paper we consider what protection means with respect to partiality and underspecification. Our treatment of protection is not meant to be exhaustive, but merely to illustrate concepts that are useful with some logics that are widely used for formal specification. (See [8, 13] for surveys that also cover additional kinds of logics that might be used in formal specification, and hence might need their own concepts of protection. Also PVS [24] represents another kind of specification logic that should be considered in extending our concepts.)

The first concept of protection we discuss is appropriate for behavioral interface specification languages (BISLs) that use a logic that accepts the existence of partial functions and has various non-classical ways to reason about them. For example, VDM-SL [18, 1] uses a logic called LPF [18, Section 3.3] [2, 3, 19], which has three logical values and two kinds of equality.[1] As another example, the specification languages COLD-K [10], uses a logic having just two logical values, but in which all other types have an improper value, $\perp$, which models the "undefined" results of partial functions, and also models computations that go into infinite loops or cause errors. In COLD-K there is also a definedness predicate, D, that allows one to reason explicitly about whether a term denotes a proper value or not. There are several other languages with similar concepts [4, 6, 26, 20, 28].

The second concept of protection we discuss is appropriate for BISLs that use a logic that does not admit the existence of partial functions, but uses *underspecification*. In such a logic, one avoids specifying a value for undefined terms [13, 17]. In this approach, to make a term "undefined" one simply does not specify its value; hence it will not be possible to prove what its value is. For example, the Larch family BISLs [14] use a mathematical component, LSL [14, Chapter 4] [15], which has this kind of

[1] However, in LPF nonstrict (i.e., strong) equality and the definedness operator, Δ, are only used in meta-arguments, since the logic is designed so that one only needs to use strict (i.e., weak) equality in proofs.

logic. (See Appendix A for more about underspecification and its relation to LSL.) The BISLs of the RESOLVE family [23] also use this kind of logic. It also seems that the draft standard for Z [16, 25], has decided to use this kind of logic [29].

It is not the purpose of this paper to advocate one kind of logic over another. Instead, this paper explores concepts of protection, with the aim of improving intuition about it and providing more guidance to specifiers. We also discuss how to prove protection from the effects of underspecification.

2 Protective Procedure Specifications

The idea of protection in a BISL was first formulated by Wing [27, Section 5.1.4]. Although we generalize that notion here, our goal is the same as Wing's original: knowing when a behavioral interface specification protects "its users from the incompleteness of the" mathematical vocabulary used in that specification "by ensuring that the meaning of the procedure specification is independent of any incompleteness" in that vocabulary (p. 123).

2.1 Partiality Protection

In a specification language like VDM-SL or COLD-K, the notion of a procedure specification that protects against partiality is relatively straightforward. This is because the associated logic explicitly includes a "bottom" element, $\perp$, and a definedness predicate, which we will write as D (where $D(\perp) = \mathit{false}$ and if x is proper then $D(x) = \mathit{true}$). The symbol $\vdash$ stands for provability in the appropriate logic (or meta-logic, if, as in LPF, the logic itself does not deal with the definedness predicate). The idea is that a specification is protective if for all possible inputs, the precondition is defined, and whenever the precondition is true, then the postcondition is defined.

Definition 2.1 (partiality-protective) *A procedure specification, S, that uses a mathematical theory, T, and has formal parameters, $\vec{x} : \vec{U}$, precondition, $Q(\vec{x})$, and postcondition, $R(\vec{x})$, is* partiality-protective *if and only if*

- $T \vdash \forall \vec{x} : \vec{U} \,.\, D(Q(\vec{x}))$, *and*
- $T \vdash \forall \vec{x} : \vec{U} \,.\, Q(\vec{x}) \Rightarrow D(R(\vec{x}))$.

For example, the VDM-SL specification of factorial in Figure 4 is partiality-protective, because the precondition is always defined, and whenever `x` satisfies the precondition, the postcondition is always defined.

2.2 Underspecification Protection

The Larch family, the RESOLVE family, and Z use logics in which all functions are total. (Since we are most familiar with Larch, we concentrate on Larch in the discussion below. The appropriate notions for RESOLVE and Z can be defined similarly.) For a logic that regards all functions as total, the notion of partiality protection has

```
fact: int -> int
fact(i) == if i = 0 then 1 else i * fact(i-1)

FACTORIAL(x: int) result: int
   pre 0 <= x and x <= 8
   post result = fact(x)
```

Figure 4: An auxiliary function specification and a protective procedure specification for factorial in VDM-SL. (Note that the factorial of 9 is larger than 2^{16}.)

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

Figure 5: A trait with an underspecified constant.

no meaning. The analogous notion, which we call "underspec-protection," is a test that the meaning of a procedure specification does not rely on underspecified terms. Note, however, that an operator may be underspecified for reasons other than being "partial." For example, in Figure 5, `bufSize` is underspecified but not partial in any sense.[2]

We define the notion of underspec-protection in three steps. First we define the notion of a primed LSL trait[3] and term. That notion is used to describe a notion of a "completely-defined" term. An LSL term is completely-defined if it can be proved to have the same value in all models of its trait. A completely-defined term is similar to a defined (non-$\perp$) term in logics like LPF; this is the main technical distinction between the two notions of protection. Finally we define the notion of underspec-protection itself.

The notion of a primed trait and term is a variation of the idea of "priming" traits and terms found in the Larch Prover (where it is used in proving that an operator is "converted" [14, pp. 142–4]).

Definition 2.2 (Primed Trait, T') *Let T be an LSL trait. Let T' be a version of the trait T with every operator f in T replaced by f', except that the following*

[2] In these logics, there is also no way to separate underspecification that is used to make operators "partial" from underspecification that is used to make specifications intentionally less constraining, as in a `choose` operator for sets.

[3] A trait is a specification of mathematical vocabulary in an augmented form of first-order logic with equality; see [14, Chapter 4] for details.

```
factTrait: trait
  includes Integer
  introduces
    fact: Int → Int
  asserts
    ∀ i: Int
      fact(0) == 1;
      (i > 0) ⇒ (fact(i) = i * fact(i-1));
```

Figure 6: A trait for factorial, written in LSL.

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 ≠), and*
- *all operators mentioned in a* **generated by** *clause.*

For example, consider the trait `factTrait`, given in Figure 6. 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` [14, p. 161]. (Operators mentioned in a **generated by** clause are meant to give a way to produce all values of a given sort; priming these would add "junk" to the specification.)

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 for primed traits. 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.

Definition 2.3 (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 for each trait T, $T \cup T' \vdash \forall x : U . x = x$. 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. As another example, the term `choose({1} ∪ {2})` is not completely-defined for the trait `ChoiceSet` (of [14, 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. The two requirements in the definition are analogous to those for partiality protection, with complete-definition tests playing the role of the definedness predicate.

```
uses factTrait(int for Int);

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

Figure 7: A specification of the factorial procedure in Larch/C++.

Definition 2.4 (underspec-protective) *A procedure specification, S, that uses trait T, has formal parameters $\vec{x} : \vec{U}$, precondition $Q(\vec{x})$, and postcondition $R(\vec{x})$, is* underspec-protective *if and only if*

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

The definition of underspec-protective suggests a direct proof technique. For example, to prove that the specification of `factorial` in Figure 7 is underspec-protective, one must show that $\mathtt{factTrait} \cup \mathtt{factTrait}'$ proves both of the following:

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

Proofs, such as the one sketched above, that a procedure specification is underspec-protective are quite tedious to carry out in detail, at least by hand.

3 Proving Underspec-Protection

In this section we describe an easier way to prove underspec-protection in a Larch family 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 specify more precisely and check what is intended to be completely-defined.

Since we are only concerned with underspec-protection in this section and the next, we will simply refer to it as "protection" in informal remarks.

3.1 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. 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" [14, p. 142]. (See Appendix B for a more complete explanation of conversion.)

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

Figure 8: An LSL trait in which `somewhatBigger` is convertible, but `somewhatBigger(i)` is not completely-defined.

```
factTraitE: trait
  includes factTrait
  implies
    exact ∀ k: Int such that k ≥ 0
      fact(k)
```

Figure 9: A trait that demonstrates the **exact** clause. The **includes** directive has the effect of textually including the trait `factTrait` given above.

However, 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 8. 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 idea is that it would allow one to make redundant claims that terms are completely-defined. For example the **exact** clause in Figure 9 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, which does not affect the trait's theory, can be used to help debug an LSL specification, by trying to prove the following property.

Definition 3.1 (provable for exact clauses) *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*

$\texttt{Exact('}x\texttt{')} = \texttt{true}$, 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 $\Rightarrow$
$\texttt{Exact('}\forall \vec{x} : \vec{T} \,.\, E\texttt{')} = \forall \vec{x} : \vec{T} \,.\, \texttt{Exact('}E\texttt{')}$
$\texttt{Exact('}\exists \vec{x} : \vec{T} \,.\, E\texttt{')} = \forall \vec{x} : \vec{T} \,.\, \texttt{Exact('}E\texttt{')}$
$\texttt{Exact('}\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3\texttt{')} = \texttt{Exact('}E_1\texttt{')}$
$\wedge\ \texttt{Exact('}E_2\texttt{')} \wedge \texttt{Exact('}E_3\texttt{')}$
$\texttt{Exact('}E\texttt{')} = \texttt{false}$, otherwise

Figure 10: Definition of `Exact`.

and $P(\vec{a})$ *is a term in the language of* T. *This clause is* provable for T *if and only if:*

$$T \cup T' \vdash \forall \vec{a} : \vec{A} \,.\, (Q(\vec{a}) \wedge Q'(\vec{a})) \Rightarrow P(\vec{a}) = P'(\vec{a}). \tag{1}$$

For example, in Figure 9, the **exact** clause is provable for `factTraitE` if the following condition is provable from $\texttt{factTraitE} \cup \texttt{factTraitE}'$.

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

The proof would proceed by induction on `k`.

3.2 Exact Predicates

For use in proving protection, we define predicates of the form $\texttt{Exact('}E\texttt{')}$, based on the form (i.e., the text) of each expression E. (These resemble the domain predicates, $\texttt{Dom('}E\texttt{')}$, described by some authors [12, 9, 5]. 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. They also resemble the definedness predicate (D) used in studies of partial algebras [7] and in COLD [10]; however D is defined model-theoretically, not syntactically.) The definition of $\texttt{Exact('}\cdot\texttt{')}$ 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 10 for the definition.[4]

For example, for the trait of Figure 9, the following holds.

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

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

3.3 Using Exact Predicates to Prove Underspec-Protection

Provided the information given in the **exact** clauses is provable 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 3.2 *Let T be a trait in which each* **exact** *clause is provable for T. Let $R(\vec{x})$ be a term with free variables, $\vec{x} : \vec{U}$. If $T \vdash \forall \vec{x} : \vec{U}$.* `Exact('`$R(\vec{x})$`')`*, then $R(\vec{x})$ is completely-defined for T.*

Proof: (by induction on the structure of terms). Suppose $T \vdash \forall \vec{x} : \vec{U}$.`Exact('`$R(\vec{x})$`')`.

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

$$T \vdash \bigwedge_{E_i(\vec{x}) \in \vec{E}(\vec{x})} \texttt{Exact('}E_i(\vec{x})\texttt{')} \wedge Q(\vec{E}(\vec{x})). \tag{2}$$

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

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

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

$$T \cup T' \vdash \forall \vec{x} : \vec{U} \; . \; Q(\vec{E}(\vec{x})) \wedge Q'(\vec{E}'(\vec{x})). \tag{4}$$

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

$$T \cup T' \vdash \forall \vec{x} : \vec{U} \; . \; E_i(\vec{x}) = E'_i(\vec{x}). \tag{5}$$

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

$$T \cup T' \vdash \forall \vec{a} : \vec{A} \, . \, (Q(\vec{a}) \wedge Q'(\vec{a})) \Rightarrow P(\vec{a}) = P'(\vec{a}). \tag{6}$$

Instantiating $\vec{a}$ to $\vec{E}(\vec{x})$, and using Formula (5), it follows that

$$T \cup T' \vdash \forall \vec{x} : \vec{U} \; . \; (Q(\vec{E}(\vec{x})) \wedge Q'(\vec{E}'(\vec{x}))) \Rightarrow P(\vec{E}(\vec{x})) = P'(\vec{E}'(\vec{x})) \tag{7}$$

But by (4), 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 5. 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,
```

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

Definition 3.3 (exact procedure specification) *A procedure specification, S, that uses trait T, has formal parameters $\vec{x} : \vec{U}$, precondition $Q(\vec{x})$, and postcondition $R(\vec{x})$, is* exact *if and only if*

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

Our suggested technique for proving protection, therefore, is to prove that the specification in question is exact.

Corollary 3.4 *Let T be a trait in which each* **exact** *clause is provable for T. Let S be a procedure specification that uses trait T. If S is exact, then S is underspec-protective.* ∎

As an example of the use of the above corollary, we show how to prove that the specification of `factorial` in Figure 7 is completely-defined with respect to the trait in Figure 9. To do this we prove that the specification is exact with respect to the trait in Figure 9. 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}
  x ≥ 0 ⇒ (Exact('result') ∧ Exact('x') ∧ x ≥ 0)
=  {by definition of Exact for variables, treating result as a variable}
  x ≥ 0 ⇒ (true ∧ true ∧ x ≥ 0)
=  {by predicate calculus}
  true
```

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.

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

Figure 11: The Larch/C++ specification of a procedure that is underspec-protective, even exact, but not deterministic.

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

Figure 12: A specification that is deterministic but not underspec-protective.

4 Discussion of Underspec-Protection

One might wonder whether a procedure specification is underspec-protective if and only if it is deterministic. However, the two notions are orthogonal. For example, the specification given in Figure 11 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 12 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.

The notion of underspec-protection should also not be confused with the specification being "well-defined". For example, the specification in Figure 13 is well-defined despite not being protective. It is well-defined because `choose`, being an operator defined in a trait, must be a mathematical function (it cannot be nondeterministic). Thus a specification that is not protective is not necessarily bad; there is no problem as long as the underspecification at the interface level is intentional.

Our technical results related to underspec-protection are summarized in Table 1. 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 [11] [14, Chapter 7]) of LSL specifications.

```
uses IntSetTrait;
int pick(IntSet s);
behavior {
  requires size(s^) > 0;
  ensures result = choose(s^) ∧ s' = delete(choose(s^), s^);
}
```

Figure 13: A specification that is well-defined but not underspec-protective. The notations `s^` and `s'` mean the starting and ending values of `s`.

Level	Facts	
Trait	exact $\Rightarrow$ completely-defined	Lemma 3.2
	completely-defined $\neq$ convertible	Figure 8
BISL	exact $\Rightarrow$ underspec-protective	Corollary 3.4
	underspec-protective $\neq$ deterministic	Figures 11 and 12
	well-defined $\not\Rightarrow$ underspec-protective	Figure 13

Table 1: Summary of results related to underspec-protection.

5 Summary and Conclusions

In this paper we have given two definitions that are instances of the concept of protection. The definition of partiality-protection can be used with languages like VDM-SL and COLD-K, since these languages use a logic that admits the existence of partial functions. Underspec-protection is an analogous notion that is necessary for languages like Larch, RESOLVE, and Z, since they use logics that deal only with total functions.

Both kinds of protection may be useful in VDM-SL or COLD-K, where one can define partial functions and use underspecification. For example, after checking that a VDM-SL specification is partiality-protective, then one could check that it was also underspec-protective (assuming that the procedure was intended to be completely specified and not underspecified). Checks that a VDM-SL procedure is underspec-protective can be done in same way as we described them for the Larch family.

Both kinds of protection may also be useful for writers of executable specifications. For example, in a language like Eiffel [22], partiality-protection for a procedure would ensure that its precondition would be flagged as false instead of encountering an error, allowing an error to happen in its body, or encountering an error in its postcondition.

Acknowledgments

Thanks to Cliff Jones, Jim Horning, Adrian Fiech, Clyde Ruby, Krishna Kishore Dhara, Matt Markland, and the anonymous referees for comments and discussions of earlier drafts. 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.

A Appendix: Understanding Underspecification in LSL

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.")

The logic used by the Larch Shared Language (LSL) [14, Chapter 4] [15] deals with partiality by using underspecification. As noted in the main body, this means that one avoids specifying a value for undefined terms, but the logic assumes that all functions are total. For example, `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 [13], the logic of LSL is classical, and thus has several pleasing formal properties. However, as Jones pointed out in a recent paper [17], there are a few subtle aspects to this kind of logic that users should be aware of.

We translate Jones's first example into the LSL trait shown in Figure 14. 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 [17].) In LSL `f(-1) = it`, because `f` has to take on some value when 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). Another way of putting Jones's point is that it is simply impossible to specify partial functions in LSL, even using recursion.

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 15. 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.

$$\mathtt{fact}(-1) == -\mathtt{fact}(-2) \tag{8}$$

This follows because `fact(-1)` denotes some (unspecified) value.

Jones's warning could have been stated more strongly, since not only is there a danger that one might specify unwanted properties, but there is also a danger that these unwanted properties might cause inconsistency. The trait `factTrait` of Figure 15, actually has quite a few such unwanted equations but manages to escape inconsistency because of special properties of the integers. (That is, the following equations are also consequences of the trait.

```
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 14: Jones's first example, a function into a one-element set.

```
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 15: Jones's factorial example.

```
fact(-1) == (-1) * fact(-2)
fact(-2) == (-2) * fact(-3)
fact(-3) == (-3) * fact(-4)
```

However, the trait is not inconsistent, because one can let `fact(`i`) == 0` for all negative integers i, which allows all these equations to be satisfied.)

To illustrate what can happen if one is not careful, consider the trait `badRecTrait` of Figure 16. At first glance, it looks like `zero` is a (silly) definition of a constant function that returns zero for any nonnegative integer. However, this specification is not careful to explicitly underspecify the value of `zero` for negative arguments. That is, although the specifier might think that it does not matter what `zero` returns for negative arguments, just ignoring the issue in the specification does mean that the value is underspecified. For example, what does the specification say about `zero(-1)`? It is easy to see that it is less than `-1`, and less than `-2`, and indeed less than any integer. But the `Integer` trait in Guttag and Horning's handbook (see [Guttag-Horning93], p. 163) does not allow there to be such an integer; so this trait is

```
badRecTrait: trait
  includes Integer
  introduces
    zero: Int → Int
  asserts
    ∀ k: Int
       zero(k) == if k = 0 then 0 else min(k, zero(k-1));
  implies
    ∀ k : Int
      zero(-1) < -1;
      zero(-1) < k;
```

Figure 16: A trait with an inconsistent recursive definition.

inconsistent, because the value of `zero(-1)` is *overspecified* — it has to satisfy too many constraints. Although none of these constraints were intended, the trait is just as inconsistent as if they were intentionally specified.

To avoid the possibility of such inconsistency arising from unintentional over-specification, it is best to use intentional underspecification. That is, to avoid the possibility that an operator may be inconsistently specified (and the need to prove that the inconsistency does not happen), it is best to use conditional equations instead of unguarded recursive equations. For example, one can writing `factTrait` as in Figure 6, where the equation for the recursive case is only postulated to hold for its intended domain [13]. By writing `factTrait` in that way, one avoids postulating Equation (8); that is, nothing at all is specified about the value of `fact(-1)`.

B 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.

B.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 [11] [14, 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 14. 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" [14, p. 142]. To prove this, one must show it for all possible arguments. The Larch Prover (LP) uses the following proof technique [14, 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 14, 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 14.)

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)`.

B.2 An extension to LSL

The **exempting** clause in the current LSL [14, Chapter 4] [15] 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 6 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 17. This form of the **exempting** clause allows one to specify the intended exemptions with an arbitrary (boolean-valued) LSL term.[5]

The extension to the LP proof technique for proving the **converts** clause in Figure 17 is simple. The **exempting** clause gives one the following formula

[5] 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**).

```
factTrait: trait
  includes Integer
  introduces
    fact: Int → Int
  asserts
    ∀ i: Int
      (i > 0) ⇒
        (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 17: A trait demonstrating the extended **exempting** clause.

```
  ∀ 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.

References

[1] D. 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, Nov. 1993. ftp://gatekeeper.dec.com/pub/standards/vdmsl/.

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

[3] J. Bicarregui, J. S. Fitgerald, P. A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide.* Springer-Verlag, New York, N.Y., 1994.

[4] A. Bijlsma. Semantics of quasi-boolean expressions. In W. H. J. Feijen et al., editors, *Beauty is Our Business*, pages 27–35. Springer-Verlag, 1990.

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

[6] A. Blikle. Three-valued predicates for software specification and validation. *Fundamenta Informaticae*, XIV:387–410, 1991.

[7] M. Broy and M. Wirsing. Partial abstract types. *Acta Informatica*, 18(1):47–64, Nov. 1982.

[8] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. C. P. Woodcock, editors, *Proceedings of the Third Refinement Workshop*, Workshops in Computing Series, pages 51–69, Berlin, 1990. Springer-Verlag.

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

[10] 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.

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

[12] S. 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, Jan. 1978.

[13] D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In J. 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.

[14] J. V. Guttag, J. J. Horning, S. Garland, K. Jones, A. Modet, and J. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, N.Y., 1993.

[15] J. V. Guttag, J. J. Horning, and A. 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, Apr. 1990. Order from src-report@src.dec.com.

[16] I. Hayes, editor. *Specification Case Studies.* International Series in Computer Science. Prentice-Hall, Inc., second edition, 1993.

[17] C. Jones. Partial functions and logics: A warning. *Inf. Process. Lett.*, 54(2):65–67, 1995.

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

[19] C. B. Jones and K. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.

[20] B. Konikowska, A. Tarlecki, and A. Blikle. A three-valued logic for software specification and validation. *Fundamenta Informaticae*, XIV:411–453, 1991.

[21] G. T. Leavens. Larch/C++ Reference Manual. Version 5.1. Available 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, Jan. 1997.

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

[23] W. F. Ogden, M. Sitaraman, B. W. Weide, and S. H. Zweben. Part I: The RESOLVE framework and discipline — a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, Oct 1994.

[24] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, Feb. 1995.

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

[26] D. S. Stefan Kahrs and A. Tarlecki. The definition of Extended ML: a gentle introduction. Technical Report ECS-LFCS-95-322, Laboratory for Foundations of Computer Science, University of Edinburgh, Oct. 1995. To appear in *Theoretical Computer Science*.

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

[28] U. Wolter, K. Didrich, F. Cornelius, M. Klar, R. Wessäly, and H. Ehrig. How to cope with the spectrum of SPECTRUM. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, volume 1009 of *Lecture Notes in Computer Science*, pages 173–189. Springer-Verlag, New York, N.Y., 1995.

[29] J. Woodcock and D. Jackson. About the semantics of partial functions in Z. Personal communication, Apr. 1996.