Enhancing the Pre- and Postcondition Technique for More Expressive Specifications

Gary T. Leavens and Albert L. Baker

TR #97-19b
September 1997, revised February, June 1999

Keywords: formal methods, liberal specification, redundancy, debugging, history constraint.
1997 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — languages, human factors; D.2.m [*Software Engineering*] Miscellaneous — reusable software; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Pre- and post-conditions, specification techniques.

To appear in Jeannette Wing and James Woodcock, editors. FM'99: World Congress on Formal Methods in Development of Computer Systems, Toulouse, France, September 1999. *Lecture Notes in Computer Science*, Copyright © Springer-Verlag, 1999.

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

Enhancing the Pre- and Postcondition Technique for More Expressive Specifications

Gary T. Leavens and Albert L. Baker

Department of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, Iowa 50011-1040 USA
http://www.cs.iastate.edu/~leavens/index.html
http://www.cs.iastate.edu/~baker/baker.html
leavens@cs.iastate.edu and baker@cs.iastate.edu
phone: +1 515 294 1580, fax: +1 515 294 1580

Abstract. We describe enhancements to the pre- and postcondition technique that help specifications convey information more effectively. Some enhancements allow one to specify redundant information that can be used in "debugging" specifications. For instance, adding examples to a specification gives redundant information that may aid some readers, and can also be used to help ensure that the specification says what is intended. Other enhancements allow improvements in frame axioms for object-oriented (OO) procedures, better treatments of exceptions and inheritance, and improved support for incompletely-specified types.

Many of these enhancements were invented by other authors, but are not widely known. They have all been integrated into Larch/C++, a Larch-style behavioral interface specification language for C++. However, such enhancements could also be used to make other specification languages more effective tools for communication.

Keywords: specification language design, expressiveness, liberal specification, redundancy, debugging, history constraint, Larch.

1 Introduction

1.1 Background and Motivation

The pre- and postcondition technique was described by Hoare in his classic article [26]. This technique forms the basis of most contemporary specification languages for sequential systems [1,15,16,18,23,28,31,41,40,42,43,47,50,51]. (However, Z [24,52] is an exception, as Z preconditions are not explicitly stated, but instead are calculated from the specification given [60, Chapter 14].)

We take as our starting point an excellent article by Jonkers [30], which, like this paper, is addressed to specification language designers. Jonkers says (page 428):

"Nowadays the pre- and postcondition technique is considered a standard technique in software development as it is being taught in almost every basic software engineering course. This gives the impression that the technique has fully matured and that it can be applied everyday in software development practice. The fact that this is not really the case is camouflaged by the sloppy and informal way pre- and postconditions are generally used in practice."

Besides reconstructing the pre- and postcondition technique, Jonkers describes several enhancements. These enhancements are found in the specification language COLD-1 [15]. The following briefly summarizes the enhancements COLD-1 makes over previous specification languages, such as VDM [1,16,28] and other languages in the Larch family [23]:

- Dependent variables, the declaration of which allows the dependent variable to be modified whenever the variables it depends on are modified. Dependent variables can be specified either directly, or indirectly using pre- and postconditions. (See also Leino's work on dependencies [37].)
- Fine-grained frame axioms using wild cards and expressions, which allow one to specify the variables that can be changed more concisely and precisely.
- Let clauses, which allow the introduction of local named abbreviations.
- Some extensions for the specification of reactive systems.

1.2 Contribution

Our work extends Jonker's work in that all the extensions we discuss in this paper are new with respect to COLD-1. Many enhancements that we describe are the work of other authors. Except for the ideas of user-selectable partial vs. total correctness, and certain forms of redundancy, it is not our intention to claim the other enhancements as our own. Instead we wish to highlight them so that they might become more widely known and used in specification language design.

We show how all these enhancements are integrated in Larch/C++ [32,33], a Larch-style behavioral interface specification language for C++. Larch/C++ adopts most of the COLD-1 extensions, except for the technical ideas for fine-grained frame axioms and the extensions for the specification of reactive systems, and includes the enhancements discussed below. This integration enhances the rhetorical effectiveness and utility of Larch/C++.

Nevertheless, the enhancements we discuss would apply equally well to other specification languages, including those outside the Larch family. That is, the ideas themselves are not specific to Larch/C++ or even to Larch, but to the pre- and postcondition technique generally.

We believe that specifications written using these enhancements provide more precise and more easily understandable contracts. Briefly, we hope that our enhancements make specifications more expressive.

By *more expressive* we mean that the specifications convey information more immediately to the reader. That is, in this paper we care not so much about what can be expressed, but how easy it is to express and understand.

It is beyond the scope of this paper to experimentally validate our hopes for increased expressiveness. Instead, we claim just to demonstrate the plausibility of increased expressiveness by showing suggestive examples, and leave for later experiential or experimental validation. What we present is a necessary first step. Furthermore, we believe that too little attention is paid to the expressiveness of specifications in the formal methods and reuse communities. We believe that it would be interesting to investigate the degree to which the expressiveness of formal methods affects their use and cost-effectiveness.

We also claim that some of the enhancements we describe can increase the quality of specifications. This is particularly true of the redundancy enhancements described in Section 5, which can be used to check that the specification says what is intended [55, 54, 56].

1.3 Overview

In Section 2 below we show how to allow the specifier to choose either total or partial correctness specifications. In Section 3, we describe a syntactic sugar, "case analysis," that helps break specifications up into more easily understood pieces. In Section 4, we describe some improvements to frame axioms. In Section 5, we describe how to add redundancy, including examples, to specifications. In Section 6, we describe "history constraints" that can constrain how states can change. Finally, we offer some conclusions.

2 Liberal Specifications

Most pre- and postcondition-based specification languages have a *total correctness* [13] semantics. That is, a specification such as Figure 1 must always terminate if the precondition is satisfied.

```
extern void inc(int& i) throw();
//@ behavior {
//@   requires assigned(i, pre) /\ i^ < INT_MAX;
//@   modifies i;
//@   ensures i' = i^ + 1;
//@ }
```

Fig. 1. The Larch/C++ specification of the C++ function `inc`.

(In Figure 1, the first line gives the C++ function's interface. It says that `inc` takes an integer argument passed by reference, returns nothing, and may not throw exceptions. The behavior of `inc` is specified in the remaining lines. The precondition starts with the keyword `requires`, and the postcondition with the keyword `ensures`. The notation `i^` is the pre-state value of the variable `i`, and `i'` is its post-state value. The notation `assigned(i, pre)` means that `i` has been assigned a proper value in the pre-state; `/\` means "and". The `modifies` clause is a frame axiom, which says that only the object `i` can have its value changed.)

A *partial correctness*, or *liberal*, semantics means that when the precondition is satisfied, then if the procedure terminates, the postcondition must hold. However, termination is not required. By termination, we mean return to the caller of a procedure, either normally or by throwing an exception. Infinite loops, jumps to other parts of the program, and program abortion are not termination.

In Larch/C++, users can specify procedures using either the total or partial correctness semantics. Specifications that use just the keyword `ensures` have a total correctness semantics, and those that use `ensures liberally` have a partial correctness semantics. (The keyword `liberally` is inspired by Dijkstra's terminology [13]; it has been suggested that `on exit` might be better.)

One use for partial correctness specifications, as in Hoare's original work [26], is to avoid finiteness issues. For example, instead of specifying `inc` as in Figure 1, one could drop the precondition conjunct `i^ < INT_MAX` and use `ensures liberally` in the postcondition. In this altered specification, the postcondition would only need to be satisfied if the procedure terminated; for example, a correct implementation could abort the program if the result could not be represented. As a contract this is less precise since no call need terminate, but it is shorter.

Such finiteness issues often arise in allocation routines, such as C++ constructors. For example, if an implementation of a constructor might plausibly need to allocate some memory from the heap, a total correctness specification would have to describe the circumstances in which there is enough memory available. Not only would such a specification be tedious and longer, but it might also overly constrain implementations. The problem is that there is no way to know how much memory all possible implementations might need.

Although one might specify that a very generous amount of memory is required for termination, doing so with just a total correctness specification would impose no obligation at all on implementations when the very generous amount was not available. In Larch/C++, one can combine total and partial correctness specifications for the same procedure, and thus more precisely specify both when a call must terminate and what must be true on termination. The semantics of such combinations uses the ideas of Dijkstra and others [13, 46, 25].

Another way out of the difficulty with allocation routines would be to change the meaning of total correctness. For example, one could use a variation on Poetzsch-Heffter's semantics [49, page 48] and require termination only if no memory allocation errors occur.

However, there are other uses for partial correctness. A prime use is in specifying when a procedure must not terminate. A simple example is the C++ `abort` procedure, which can be specified as in Figure 2. This procedure can always be called, but when called must abort program execution instead of terminating, and hence cannot be specified with a total correctness semantics.

```
void abort();
//@ behavior {
//@   ensures liberally false;
//@ }
```

Fig. 2. Specification of `abort`.

The use of partial correctness, together with case analysis (see below), allows one to specify exactly under what conditions a procedure must not terminate. This technique is useful in precisely specifying contracts for procedures written for languages (or compilers) without exception handling. This idea appears in sugared form in the LCL `checks` clause [23,55,54,56].

Partial correctness is also useful for specifying procedures for which there is no known totally-correct implementation. Interpreters for Turing-complete languages are examples.

3 Case Analysis

A simple syntactic sugar, which we call *case analysis*, is helpful in breaking up specifications into more manageable chunks, and in specifying procedures that can throw exceptions. Its advantage over special-purpose notations for exceptions (as in LM3 [29,23], to cite just one example) is that it is also useful for other kinds of case analysis This sugar was pioneered by Wing [59, Section 4.1.4]. The idea is that a specification can be split into several cases, all of which must be satisfied by a correct implementation. This concept was independently reinvented by Wills [57]. Wills called specification cases "capsules", and used them effectively in OO specifications.

In Larch/C++, specification cases are separated by the keyword `also`. Consider the example of Figure 3. This example shows a specification with two cases. The first case specifies an exception, the second the function's "normal" behavior, which is to set each element of the argument array to zero. (The notation `\A` means "for all".)

The desugaring of a specification with case analysis turns it into a specification with a single total correctness and a single partial correctness case. Each such desugared case has as its precondition the disjunction (written `\/`) of the preconditions of each corresponding case, and as its postcondition a conjunction of implications, with each precondition implying (written `=>`) the corresponding postcondition. For example, the specification in Figure 4 is the desugaring of

```
#include "BadSize.h"
extern void ZeroArray(double x[], int n) throw(BadSize);
//@ behavior {
//@    requires n <= 0;
//@    ensures throws(BadSize);
//@  also
//@    requires 0 < n /\ n <= size(x) /\ allocated(x, pre);
//@    modifies x;
//@    ensures returns
//@         /\ (\A i: int ((0 <= i /\ i < n) => x'[i] = 0.0));
//@ }
```

Fig. 3. Specification of the C++ function `ZeroArray`. The predicate `throws(BadSize)` is true when the function terminates and throws the named exception; `returns` is true when the function terminates normally. The predicate `allocated(x, pre)` is true when `x` is allocated in the pre-state.

the specification in Figure 3. We think that Figure 3 is significantly easier to understand.

```
#include "BadSize.h"
extern void ZeroArray(double x[], int n) throw(BadSize);
//@ behavior {
//@   requires n <= 0 \/ (0 < n /\ n <= size(x) /\ allocated(x, pre));
//@   modifies x;
//@   ensures ((n <= 0) => (throws(BadSize) /\ unchanged(x)))
//@        /\ ((0 < n /\ n <= size(x) /\ allocated(x, pre))
//@            => (returns
//@                /\ \A i: int ((0 <= i /\ i < n) => x'[i] = 0.0)));
//@ }
```

Fig. 4. Desugared specification of `ZeroArray`.

The interaction of frame axioms with this desugaring is subtle. The frame for the desugared specification has to allow all modifications permitted in each original case, since that permission is needed by the whole procedure. To keep the original meaning, however, the operator `unchanged` is used as needed in each case. For example, in Figure 4, `unchanged(x)` is conjoined to the original first case's postcondition.

With just this sugar, however, precondition conjuncts that are shared among cases would have to be repeated in each case. To avoid such repetition, cases in Larch/C++ can be put in the scope of a precondition (and can also be nested). For example, in Figure 5, the precondition **`assigned(s, pre)`** applies to both cases. The desugaring first conjoins the outer precondition to each of the inner ones, and applies the previous desugaring. Extracting common parts of preconditions like this also highlights them for the reader. (We attach no special semantics to such common preconditions, unlike Poetzsch-Heffter [49, pages 96-97].)

```
#include "Stack.h"
#include "BadSize.h"
extern void pop2(Stack & s) throw(BadSize);
//@ behavior {
//@   requires assigned(s, pre);
//@   {
//@      requires size(s^) < 2;
//@      ensures throws(BadSize);
//@    also
//@      requires size(s^) >= 2;
//@      modifies s;
//@      ensures returns /\ s' = pop(pop(s));
//@      ensures redundantly size(s') = size(s^) - 2;
//@   }
//@ }
```

Fig. 5. Specification of `pop2`. The `ensures redundantly` clause is explained below.

For OO specification languages, Wills pointed out that one can understand inheritance of specifications as meaning that subtype objects must satisfy the cases specified for them explicitly, as well as those of their supertypes. This ensures that subtyping is behavioral [11,42]; that is, subtype objects can be reused according to their supertypes' contracts.

4 Framing

A frame axiom in a procedure specification says that "nothing else changes" [5]. VDM and Z both have features to permit the specification of frame axioms (write permissions in VDM, and Δ in Z). In the Larch family, interface specifications languages have followed Wing's design for Larch/CLU [58] in using the `modifies` clause to say that only the objects listed may have their abstract values changed.

In Larch/C++, the meaning of the `modifies` clause "`modifies i;`" is translated by a predicate like the following (see [33, Section 6.2.3.4] for exact details), which can be thought of as conjoined to the postcondition.

```
  ModifiedObjects(pre, post) \subseteq {i, residue_i}
```

In the above, the term `ModifiedObjects(pre, post)` denotes the set of all objects modified in the transition from the pre-state to the post-state, and `\subseteq` is a subset operator. The object `residue_i` stands for whatever objects `i` may depend on that are not in scope [37, Section 11.3]. The `modifies` clause gives considerable notational abbreviation, because it asserts that all objects not mentioned retain their values.

4.1 Trashing

In the Larch family, predicates use the logic of the Larch Shared Language, which is a logic of total functions [21, 35]. In such a logic, the pre- and post-states, which are modeled by functions, will return proper values for objects that are not allocated or that are not assigned a proper value. To avoid ill-defined specifications, it is important that a specification written in such a logic ensures that whenever an object's value is mentioned in a given state, the object is allocated (i.e., found in the domain of the state function), and assigned (i.e., given a proper value). If this is not done, then logical problems may occur [8, 27, 36].

To avoid such problems in the semantics of the `modifies` clause, the set `ModifiedObjects(pre, post)` can only include objects that are assigned values in both the pre- and post-states and change their values, or that are allocated in the pre-state and become assigned in the post-state.

However, in C++ and other languages without garbage collection, procedures can *trash* an object, either by deallocating it or by making it unassigned (for example, by "uninitializing" it from an unassigned variable). Since these actions are not considered modifications, they are not covered by the `modifies` clause. However, without additional support from the specification language, specifiers would have to make assertions about which objects remain allocated and assigned in each postcondition [7], which would be inconvenient and verbose.

To avoid having users write in postconditions assertions about what is not trashed, Chalin [7] argued for a second part to the frame axiom in Larch interface specifications. In Larch/C++ this is called the `trashes` clause. Only the objects listed in the `trashes` clause may be trashed; hence all objects not mentioned must remain assigned and allocated if they were in the pre-state, and an omitted `trashes` clause means that nothing may be trashed.

As with the `modifies` clause, the `trashes` clause is a permission, not a requirement to trash the objects mentioned. Consider the example in Figure 6 [33, Section 6.3.2.1]. The object pointed to by `cp` may be trashed, since it is mentioned in the `trashes` clause. The postcondition says that it must be trashed when the value of `ref_count` drops to 0, but may not be otherwise.

In Larch/C++, the meaning of the `trashes` clause "`trashes *cp;`" is translated by a predicate like the following (see [33, Section 6.2.3.4] for details), which can be thought of as conjoined to the postcondition.

```
  TrashedObjects(pre, post) \subseteq {*cp, residue_star_cp}
```

```
extern void dec_ref(char *cp, int & ref_count) throw();
//@ behavior {
//@   requires allocated(cp, pre) /\ assigned(ref_count, pre)
//@       /\ ref_count^ >= 1;
//@   modifies ref_count;
//@   trashes *cp;
//@   ensures ref_count' = ref_count^ - 1
//@        /\ (if ref_count' = 0 then trashed(*cp) else ~trashed(*cp));
//@   ensures redundantly ref_count^ > 1 => ~trashed(*cp);
//@   example ref_count^ = 1 /\ ref_count' = 0 /\ trashed(*cp);
//@ }
```

Fig. 6. Specification of the C++ function `dec_ref`. The `ensures redundantly` and `example` clauses are explained below.

As above, the object `residue_star_cp` stands for whatever objects `*cp` may depend on that are not in scope [37, Section 11.3].

5 Redundancy

A *redundant* part of a specification does not itself form part of the contract, but instead is a formalized commentary on it. By allowing a specifier to state redundant properties explicitly, a specification language becomes more expressive. First, it allows specifiers to state properties that are important for readers, without cluttering up the main parts of the specification. More importantly, redundant parts, since they are marked as redundant, allow checking of the main parts of the specification. One important benefit is that the reader can check his or her understanding of the main parts against the redundant parts. Another benefit is that the specifier can record more of the thinking that went into the specification; for example, various examples or properties of the specification may be thought of first, and these do not have to be dropped when a more general form is discovered.

The Larch family has emphasized the benefit of checking how well a specification captures the specifier's intuition by comparing the redundant parts against the main parts; such checking is called "debugging" a specification [17]. For example, the Larch Shared Language incorporates features that can be used to state redundant claims about theories [23, Chapter 7].

5.1 Redundant Postconditions

Tan's work on LCL introduced redundancy into a specification language with pre- and postconditions [55, 54, 56]. Of particular relevance here are Tan's "procedure claims," which state redundant properties that follow from the main part of a specification. In Larch/C++, one can use an **`ensures redundantly`** clause to state procedure claims. For example, in Figure 5 the **`ensures redundantly`** clause in the second specification case highlights a property of that case; it says that the stack's size decreases by two. Another example occurs in Fig 6.

To use redundant postconditions in debugging a specification, for each such redundancy claim, one would try to prove the following, where *Pre* is the case's precondition, *Frame* is the predicate that translates its frame axioms, *Post* is its postcondition, and *RedunPost* is the claimed redundant postcondition [55, 54, 56] [33, Section 6.8]. (All of these should be in their desugared forms.)

$$\mathit{Pre} \wedge \mathit{Frame} \wedge \mathit{Post} \Rightarrow \mathit{RedunPost} \tag{1}$$

5.2 Examples

When we give problem statements to students, we observe that many students primarily focus on examples. By adding examples as another form of redundancy to specifications one gains the benefits of additional redundancy as well as the ability to convey more clearly what is to be done. (Examples as part of interface specifications first appeared in Larch/C++ [32].) For instance, in Figure 7, examples are used to show that **`isqrt`** is underspecified; the two examples given show different approximations that may be returned for the square root of 31.

```
extern unsigned int isqrt(unsigned int & x) throw();
//@ behavior {
//@   requires assigned(x, pre);
//@   ensures (result-1)*(result-1) < x^ /\ x^ < (result+1)*(result+1);
//@   example x^ = 31 /\ result = 6;
//@   example x^ = 31 /\ result = 5;
//@ }
```

Fig. 7. Specification of the C++ function `isqrt`.

One might wonder whether examples are needed when one has case analysis; for example, why not specify **`isqrt`** as in Figure 8? One reason is that this style of specifying examples would not mark the examples as redundant for the reader. Worse, the specification in Figure 8 is inconsistent, because it says that when **`x`** is 31, the result must be both 5 and 6.

Examples can also be used to help debug specifications. What should be checked is that an example, together with the frame, describes a pair of states that are in the relation specified by the specification's main parts. In terms of predicates, this means that for each example, one should prove the following,

```
extern unsigned int isqrt(unsigned int & x) throw();
//@ behavior {
//@  requires assigned(x, pre);
//@  {
//@    ensures (result-1)*(result-1) < x^ /\ x^ < (result+1)*(result+1);
//@  also
//@    requires x^ = 31;
//@    ensures result = 6;
//@  also
//@    requires x^ = 31;
//@    ensures result = 5;
//@  }
//@ }
```

Fig. 8. A bad (inconsistent) specification of `isqrt`; this shows how examples are different than specification cases.

where *Example* is the example predicate, and *Pre*, *Frame*, and *Post* are as before.

$$(\mathit{Example} \wedge \mathit{Frame}) \Rightarrow (\mathit{Pre} \Rightarrow (\mathit{Frame} \wedge \mathit{Post})) \qquad (2)$$

By predicate calculus, this is the same as the following.

$$(\mathit{Example} \wedge \mathit{Frame} \wedge \mathit{Pre}) \Rightarrow \mathit{Post} \qquad (3)$$

We believe that it is best to give examples that do not contradict the precondition of a specification; hence it is also worthwhile to check that the conjunction of the example predicate, frame, and precondition is consistent.

The reason why the frame is conjoined to the example predicate in Formula 2 is to avoid forcing the specifier to state what objects are not modified in examples. For instance, in Figure 7, if the frame axiom were not conjoined to the example predicate, then there would be no way to prove that the example and the precondition imply the frame and the postcondition for that example, since the example predicate says nothing about the value of `x` in the post-state.

5.3 Redundant Preconditions

One can also apply the idea of redundancy to the precondition. The **`requires redundantly`** clause in Larch/C++ is the analog of the **`ensures redundantly`** clause for the precondition. It allows one to state redundant preconditions. Redundant preconditions are sometimes useful for pointing out to the reader properties that follow from the semantics of the specification language, such as that certain objects are allocated or assigned. For example, in Figure 9, the **`requires redundantly`** clause highlights the fact that reference arguments are implicitly required to be allocated, and that unsigned integers are non-negative.

```
extern unsigned int isqrt(unsigned int & x) throw();
//@ behavior {
//@   requires assigned(x, pre);
//@   requires redundantly allocated(x, pre) /\ x^ >= 0;
//@   ensures (result-1)*(result-1) < x^ /\ x^ < (result+1)*(result+1);
//@   example x^ = 31 /\ result = 6;
//@   example x^ = 31 /\ result = 5;
//@ }
```

Fig. 9. A specification of `isqrt` that shows the use of `requires redundantly`.

To use the **`requires redundantly`** clause in debugging a specification, one would prove the following, where again *Pre* is the desugared precondition, and *RedunPre* is the redundant precondition.

$$Pre \Rightarrow RedunPre \tag{4}$$

It would be possible to have an analog of the **`example`** clause for preconditions, say with an **`example input`** clause. The example input predicates would be used in debugging the specification by checking that they are consistent with the precondition. Example inputs are not included in the current version of Larch/C++ [33], because we have not found a great need for them.

5.4 Redundant Frames

Larch/C++ was also the first interface specification language to extend the idea of redundancy to the **`modifies`** and **`trashes`** clauses. In Larch/C++, one can use **`modifies redundantly`** and **`trashes redundantly`** clauses. One use for such clauses is to highlight objects that are implicitly allowed to be modified or trashed because some explicitly named object has been declared to depend on them [37]. The debugging of redundant frames is analogous to that used for redundant preconditions; that is, one would prove that the permissions that are claimed to be redundant follow from the language's semantics and the explicit permissions.

5.5 An Alternative Design for Redundancy

We now briefly describe an alternative design for redundancy that has been considered for Larch/C++, but never adopted. We are experimenting with it in our specification language for Java [34], and it may be of interest to other specification language designers.

The idea is that instead of having clauses that allow the specification of redundancy, that one label entire specification cases as redundant or examples. For example, one might write the specification of Figure 6 as in Figure 10.

```
extern void dec_ref(char *cp, int & ref_count) throw();
//@ behavior {
//@   requires allocated(cp, pre) /\ assigned(ref_count, pre)
//@       /\ ref_count^ >= 1;
//@   modifies ref_count;
//@   trashes *cp;
//@   ensures ref_count' = ref_count^ - 1
//@        /\ (if ref_count' = 0 then trashed(*cp) else ~trashed(*cp));
//@ }
//@ behavior redundantly {
//@   requires allocated(cp, pre) /\ assigned(ref_count, pre)
//@       /\ ref_count^ > 0;
//@   modifies ref_count;
//@   trashes *cp;
//@   ensures ref_count^ > 1 => ~trashed(*cp);
//@ }
//@ example {
//@   requires ref_count^ = 1;
//@   modifies ref_count;
//@   trashes *cp
//@   ensures ref_count' = 0 /\ trashed(*cp);
//@ }
```

Fig. 10. An alternative style for writing redundancy into specifications. This is not part of Larch/C++, but given in a Larch/C++ style.

One advantage of this style is that it more cleanly separates the redundant parts of a specification from the main parts. Also, examples seem clearer, because the descriptions of the pre- and post-states are separated into the requires and ensures clauses of the example.

The disadvantage of this style is that the specifications become somewhat more verbose. In a `behavior redundantly` clause, one must repeat the precondition and frame, which is not necessary with `ensures redundantly`. While an `example` clause does not need to repeat the precondition, it does seem necessary to repeat the frame in examples, because this keeps the semantics of an omitted `modifies` or `trashes` clause uniform. However, there might be ways of making this more palatable.

6 History Constraints

Many specification languages allow one to state invariants for the values of an abstract data type (ADT). An invariant property is one that must be true of each object of the ADT in all visible states. A *visible* state is one that can be observed by clients of the ADT. Such invariants can be seen as an expressive way to state properties that would otherwise have to be repeated in every operation's pre- and postcondition. However, invariants are not mere notational abbreviations, because they apply to all operations, even when new ones are added to an ADT.

Liskov and Wing introduced a similar idea as an aid to specifying OO programs that use behavioral subtyping [39, 38]. A *history constraint* for a type describes a property of objects of that type (and all subtypes) that must hold for any ordered pair of visible states in a computation, where the first state occurs before the second. To make sense, such a property must describe a reflexive and transitive relation on states. History constraints, if not stated as such, would otherwise have to be repeated in every operation's postcondition. However, history constraints are not mere notational abbreviations, because they apply to all operations, even new ones added in subtypes.

A simple example is the constraint that some field of an object never changes its value, once initialized. For instance, in the specification of a `BoundedStack` class in Larch/C++, one might write the following history constraint, to state that a Stack's field `max_size` never changes.

```
//@ constraint max_size^ = max_size';
```

The `max_size` field is allowed to be initialized, because history constraints do not apply to constructors, as the pre-state value of the object is not visible. (Technically, in Larch/C++ this is because the field has not yet been assigned a proper value upon entry to a constructor.) For analogous reasons history constraints do not apply to destructors. However, the example constraint does say that one *cannot* list `make_size` in a `modifies` clause for a normal operation (C++ member function) of the type `BoundedStack`. It thus collects information that would otherwise be spread out in all the `modifies` clauses of all the operations. Furthermore, the immutability of a field like this would only be written negatively, by not being listed in all these `modifies` clauses. Finally, the immutability of a field could be changed by new operations or by subtypes if it were not listed in the history constraint.

History constraints can also be used to succinctly express monotonic relationships between pre- and post-states. For example, the Larch/C++ manual's specification of a class `Person` [33, Section 7.1.1], includes the following history constraint, which expresses the inexorable arrow of time.

```
//@ constraint age^ <= age';
```

To allow debugging of invariants and history constraints, Larch/C++ also allows one to state redundant invariants and history constraints, using **`invariant redundantly`** and **`constraint redundantly`** clauses.

An innovation in Larch/C++ is that one can limit a history constraint so that it only applies to various named operations [11] [33, Section 7.4]. This can be used to collect common, monotonic, parts of the postconditions of several operations in one place. A more general version of this idea was advocated by Borgida *et al.* as an approach to dealing with frame axioms [5]. The form found in Larch/C++ is useful in specifying history constraints for types that are intended as supertypes of weak behavioral subtypes [11, 10] [33, Section 7.8]. However, an explanation of weak behavioral subtyping is outside the scope of this paper.

7 Other Related Work

Our goal of making pre- and postcondition specifications more expressive is also served by the refinement calculus [2–4, 43–45]. The major extension in the refinement calculus is the use of abstract programs as specifications. These are programs that may include specification statements (and other kinds of nonconstructive statements). This makes it possible to specify higher-order procedures conveniently, and is particularly useful in component-based or event-driven settings [6]. However, this extension is orthogonal to the techniques we have discussed.

The work of Perry on Inscape [48] also has as one of its goals making pre- and postcondition specifications more practical. It adds to postconditions the notion of an obligation, which clients are expected to satisfy eventually. Again, this extension is orthogonal to those discussed in this paper. Inscape also splits preconditions up into three kinds, although none of them are redundant and thus cannot be used for debugging specifications. Perry's Instress tool uses static analysis to help debug programs, not specifications.

The Extended Static Checker from Compaq SRC [9] carries on this tradition of static analysis using specifications to help debug programs; again the work is not aimed at helping debug specifications. The specifications used in this checker do, however, have some additional constructs for more expressive framing than what is described in this paper.

Our emphasis on expressiveness in specifications can be seen as following the emphasis on expressive notation in the "calculational school" of Dijkstra, Gries, and others (see, e.g., [12, 14, 19, 20]). These authors have considerably adapted standard mathematical notations to be more consistent and communicative. However, they have not directed much attention to the pre- and postcondition technique itself. Similarly, the specification language Z has a great variety of notational refinements, but these refinements are not aimed at the pre- and postcondition technique.

8 Conclusions

In this paper we have described several enhancements to the pre- and postcondition technique for specifications. These enhancements contribute to the expressiveness of Larch/C++, and could be adapted to other specification languages. We have suggested how the enhancements help the specifier communicate more effectively with potential clients and implementors. Moreover, they do not result in any loss of formal rigor.

In our experience, the most significant of these enhancements is the ability to add redundant examples to specifications. In addition to their potential use in debugging specifications, we have found that they can help make specifications clearer. We are also excited about their potential for automated testing [22].

Besides examples, the enhancement we use most often is case analysis [59, Section 4.1.4] [57]. This is helpful in stating specifications of procedures that may throw exceptions. However, since it is more general than a special-purposed notation for exceptions, it is also useful in breaking up the logic of a specification into more easily understood parts.

Even if specification language designers do not like our syntax, we hope they will address the issues we have raised and go beyond them. We also look forward to experimental tests of the expressiveness of these enhancements, and the eventual refinement of our ideas by that research.

Acknowledgments

Thanks to Yoonsik Cheon, Krishna Kishore Dhara, Matt Markland, and Clyde Ruby for their work on Larch/C++. Thanks to Patrice Chalin, Peter Müller, and Rustan Leino for several discussions about the semantics of Larch/C++. Thanks to Kishore, Peter, Rustan, and Arnd Poetzsch-Heffter, for many helpful suggestions about an earlier draft of this paper.

The work of both authors was supported in part by the National Science Foundation under Grant CCR-9803843. Leavens's work was also supported in part under Grant CCR-9503168.

References

1. Derek Andrews. *A Theory and Practice of Program Development.* FACIT. Springer-Verlag, London, UK, 1997.
2. R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, August 1988.
3. R. J. R. Back and J. von Wright. Combining angels, deamons and miracles in program specifications. *Theoretical Computer Science*, 100(2):365–383, June 1992.
4. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.
5. Alex Borgida, John Mylopoulos, and Raymond Reiter. '... and nothing else changes': The frame problem in procedure specification. In *Proceedings Fifteenth International Conference on Software Engineering, Baltimore*, May 1993. Preliminary version obtained from the authors.
6. Martin Büchi and Emil Sekerinski. Formal methods for component software: The refinement calculus perspective. In *Proceedings of the Second Workshop on Component-Oriented Programming (WCOP)*, June 1997. ftp://ftp.abo.fi/pub/cs/papers/mbuechi/FMforCS.ps.gz.
7. Patrice Chalin. *On the Language Design and Semantic Foundation of LCL, a Larch/C Interface Specification Language.* PhD thesis, Concordia University, 1455 de Maisonneuve Blvd. West, Montreal, Quebec, Canada, October 1995. Available as CU/DCS TR 95-12, from the URL ftp://ftp.cs.concordia.ca/pub/chalin/tr.ps.Z.

8. Patrice Chalin, Peter Grogono, and T. Radhakrishnan. Identification of and solutions to shortcomings of LCL, a Larch/C interface specification language. In Marie-Claude Gaudel and James Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 385–404, New York, N.Y., March 1996. Springer-Verlag.
9. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, Dec 1998.
10. Krishna Kishore Dhara. Behavioral subtyping in object-oriented languages. Technical Report TR97-09, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames IA 50011-1040, May 1997. The author's Ph.D. dissertation.
11. Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.
12. E. W. Dijkstra, editor. *Formal Development of Programs and Proofs.* University of Texas at Austin Year of Programming series. Addison-Wesley Publishing Co., 1990.
13. Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976.
14. Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and program semantics.* Springer-Verlag, NY, 1990.
15. 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.
16. John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools in Software Development.* Cambridge, Cambridge, UK, 1998.
17. 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.
18. M. Gogolla, S. Conrad, G. Denker, R. Herzig, N. Vlachantonis, and H. Ehrig. TROLL *light* — the language and its development environment. In Manfred Broy and Stefan Jähnichen, editors, *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, volume 1009 of *Lecture Notes in Computer Science*, pages 205–220. Springer-Verlag, New York, N.Y., 1995.
19. David Gries. Teaching calculation and discrimination: A more effective curriculum. *Communications of the ACM*, 34(3):44–55, March 1991.
20. 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.
21. 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.
22. M. Gurski and A. L. Baker. Testing SPECS-C++: A first step in validating distributed systems. In *Intellegent Information Management Systems*, pages 105–108, Anaheim, 1994. The International Society for Mini and Microcomputers - ISMM.
23. 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.

24. I. Hayes, editor. *Specification Case Studies.* International Series in Computer Science. Prentice-Hall, Inc., second edition, 1993.
25. Wim H. Hesselink. *Programs, Recursion, and Unbounded Choice*, volume 27 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, New York, N.Y., 1992.
26. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.
27. C.B. Jones. Partial functions and logics: A warning. *Information Processing Letters*, 54(2):65–67, 1995.
28. Cliff B. Jones. *Systematic Software Development Using VDM.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.
29. Kevin D. Jones. LM3: A Larch interface language for Modula-3: A definition and introduction: Version 1.0. Technical Report 72, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue Palo Alto, CA 94301, June 1991. Order from src-report@src.dec.com.
30. H. B. M. Jonkers. Upgrading the pre- and postcondition technique. In S. Prehn and W. J. Toetenel, editors, *VDM '91 Formal Software Development Methods 4th International Symposium of VDM Europe Noordwijkerhout, The Netherlands, Volume 1: Conference Contributions*, volume 551 of *Lecture Notes in Computer Science*, pages 428–456. Springer-Verlag, New York, N.Y., October 1991.
31. Kevin Lano. *The B Language and Method: A guide to Practical Formal Development.* Formal Appoaches to Computing and Information Technology. Springer-Verlag, London, UK, 1996.
32. Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Haim Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, Boston, 1996. An extended version is TR #96-01d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011.
33. Gary T. Leavens. Larch/C++ Reference Manual. Version 5.41. 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, April 1999.
34. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06e, Iowa State University, Department of Computer Science, June 1999.
35. Gary T. Leavens and Jeannette M. Wing. Protective interface specifications. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214 of *Lecture Notes in Computer Science*, pages 520–534. Springer-Verlag, New York, N.Y., 1997.
36. Gary T. Leavens and Jeannette M. Wing. Protective interface specifications. *Formal Aspects of Computing*, 10:59–75, 1998.
37. K. Rustan M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.
38. Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
39. Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices*, 28(10):16–28, October 1993. *OOPSLA '93 Proceedings*, Andreas Paepcke (editor).

40. David Luckham. *Programming with Specifications: An Introduction to Anna, A Language for Specifying Ada Programs.* Texts and Monographs in Computer Science. Springer-Verlag, New York, N.Y., 1990.
41. David Luckham and Friedrich W. von Henke. An overview of anna - a specification language for Ada. *IEEE Software*, 2(2):9–23, March 1985.
42. Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, N.Y., second edition, 1997.
43. Carroll Morgan. *Programming from Specifications: Second Edition.* Prentice Hall International, Hempstead, UK, 1994.
44. Carroll Morgan and Trevor Vickers, editors. *On the refinement calculus.* Formal approaches of computing and information technology series. Springer-Verlag, New York, N.Y., 1994.
45. Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.
46. Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.
47. William F. Ogden, Murali Sitaraman, Bruce W. Weide, and Stuart H. Zweben. Part I: The RESOLVE framework and discipline — a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, Oct 1994.
48. D. E. Perry. The Inscape environment. In *Proceedings of the 11th International Conference on Software Engineering*, pages 2–12, May 1989.
49. Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.
50. David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.
51. Murali Sitaraman, Lonnie R. Welch, and Douglas E. Harms. On specification of reusable software components. *International Journal of Software Engineering and Knowledege Engineering*, 3(2):207–229, 1993.
52. J. Michael Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, New York, N.Y., second edition, 1992.
53. Susan Stepney, Rosalind Barden, and David Cooper, editors. *Object Orientation in Z.* Workshops in Computing. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.
54. Yang Meng Tan. Formal specification techniques for promoting software modularity, enhancing documentation, and testing specifications. Technical Report 619, Massachusetts Institute of Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, Mass., June 1994.
55. Yang Meng Tan. Interface language for supporting programming styles. *ACM SIGPLAN Notices*, 29(8):74–83, August 1994. Proceedings of the Workshop on Interface Definition Languages.
56. Yang Meng Tan. *Formal Specification Techniques for Engineering Modular C Programs*, volume 1 of *Kluwer International Series in Software Engineering.* Kluwer Academic Publishers, Boston, 1995.
57. Alan Wills. Specification in Fresco. In Stepney et al. [53], chapter 11, pages 127–135.
58. Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.
59. Jeannette Marie Wing. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.
60. Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall International Series in Computer Science, 1996.