On the Frame Problem in Procedure Specifications[1]

Alex Borgida [2]
Rutgers University

John Mylopoulos [3]
University of Toronto

Raymond Reiter [3]
University of Toronto and
the Canadian Institute for Advanced Research

Abstract

We give examples of situations where formal specifications of procedures in the standard pre/postcondition style become lengthy, cumbersome and difficult to change, a problem which is particularly acute in the case of object-oriented specifications with inheritance. We identify the problem as the inability to express that a procedure changes *only* those things it has to, leaving everything else unmodified, and review some attempts at dealing with this "frame problem" in the Software Specification community.

The second part of the paper adapts a recent proposal for a solution to the frame problem in Artificial Intelligence --- the notion of explanation closure axioms --- to provide an approach whereby one can state such conditions succinctly and modularly, with the added advantage of having the specifier be reminded of things that she may have omitted saying in procedure specifications. Since this approach is based on standard Predicate Logic, its semantics is relatively straight-forward. The paper also suggests an algorithm which generates syntactically the explanation closure axioms from the pre/postcondition specifications, provided they are written in a restricted language, and suggests a model theory supporting it.

Keywords: formal specifications, formal specification languages, proof obligations, semantics of specification languages, inheritance.

---

[1] This paper is a modified and extended version of the paper titled "...And Nothing Else Changes: The Frame Problem in Procedure Specifications" presented by the authors at the Fifteenth International Conference on Software Engineering, Baltimore, May 1993. This research was supported by grants from the National Science and Engineering Research Council of Canada and the Institute for Robotics and Intelligent Systems (IRIS), funded by the Government of Canada through the Networks of Centres of Excellence programme.

[2] Author's full address: Department of Computer Science, Rutgers University , New Brunswick, NJ 08904 , USA.

[3] Author's full address: Department of Computer Science, University of Toronto, 6 King's College Road, Toronto, Ontario M5S 1A4, Canada.

1. Introduction

The arguments for preparing first a formal specification of a program, before implementing it in some programming language, are by now well known and widely accepted. Program specifications present *what* the desired program is supposed to do rather than *how* it is going to accomplish it, using some suitable notation, which we shall call a *formal specification language* (or, just *specification language*). The adequacy of such language is determined by general criteria such as expressiveness, but also by

- **notational suitability**, i.e., the degree to which a specification language makes it possible for the specifier to express the intent of a procedure in a *precise* yet *simple*, *concise, understandable*, *modular* and *easily modifiable* way;

- **capacity to support formal treatment,** i.e., the extent to which a formal specification language provides a foundation and lends support to a methodology for formally proving properties about the program and its development.

The first criterion focuses on human engineering concerns, in contrast to its expressiveness. The importance of such concerns is supported by much of the research on formal specification languages. After all, one could state everything that is needed for a formal specification using the notations of a sufficiently rich set theory or predicate logic, and if it was only an issue of expressive power, there would be no need for VDM, Z, Larch, etc., nor for notions such as object-orientation.

The second criterion measures the degree to which formality leads to a methodology, preferably supported by tools, for proving properties about specifications. One familiar kind of proof involving specifications is to show that some implementation meets it. Another kind of "proof obligation", which arises in the use of model-based specification techniques such as VDM and Z, requires the specifier to show that each procedure maintains the global program-state invariants that are part of the specification (e.g., see [Cohen 86]). State invariants (for example, *"Only students who have paid their registration fees can appear on a class list"*) are also known as integrity constraints in the world of databases. Such constraints are usually stated as part of the database definition, independently of procedures that access or update the database, because the database persists between runs of procedures and because system evolution often requires new procedures to be added. If a procedure `p` is specified in terms of a pair of assertions `(pre-p, post-p)` -- referring to the conditions required to hold before the invocation of `p`, and those required to hold once `p` terminates -- and if `I` is an invariance assertion (involving a single program state), then this proof obligation usually has the form

$$I_{before} \wedge \text{pre-p} \wedge \text{post-p} \Rightarrow I_{after}$$

where I_{before} evaluates the invariant `I` with respect to the program state before execution of procedure `p` (the state in which `pre-p` is evaluated) and I_{after} with respect to the state in which `post-p` is evaluated. We'll have the opportunity to discuss these proof obligations in a more formal setting later in the paper.

The first aim of this paper is to make the reader aware of a family of problems which arise in formal specifications using the pre/postcondition notation, and which is related to a long-standing problem in the field of Artificial Intelligence (hereafter AI), called the *frame problem* [McCarthy69]. We present examples illustrating this problem, which becomes a serious impediment for large object-oriented specifications where *inheritance* plays a central role. The examples are intended to demonstrate that failure to deal with the frame problem compromises a formal specification language with respect to its notational suitability and its capacity to support a methodology for formally proving properties of specifications. The rest of the paper discusses some ways in which existing specification languages endeavor to cope with the problem and then presents a novel approach, based on recent work intended to solve the frame problem in planning applications within AI.

2. The Frame Problem: Theme and Variations

Since the claims in the sequel apply equally well to any specification language based on pre/postcondition assertions, we have used the more neutral notation of (First Order) Predicate Logic to express our examples. However, in order to establish this claim of generality, we will repeatedly also show specifications in the notation of one of the more familiar specification languages, such as VDM or Z.

2.1 The Frame Problem

Consider part of the specification of a simple procedure *enrolInCourse*, which records the enrollment of a student `st` in a course `crs`.

enrolInCourse(st,crs)

```
PRE: size(crs) < classLimit(crs)   ∧  ¬EnrolledIn(st,crs)

POST: size'(crs) = size(crs) + 1   ∧   EnrolledIn'(st,crs)
```

Here `size` and `classLimit` are functions, while `EnrolledIn` is a predicate[4]. To refer to the values of the variables immediately before and after the execution of the procedure we adopt the usual unprimed/primed notation. The above specification then declares as precondition to enrolling in a course the availability of places (i.e., that the number of students already enrolled in the course is less than the class limit) and that the particular student `st` hasn't already enrolled in the particular course `crs`[5]. The postcondition, on the other hand, specifies the effect of the procedure on the final state, including incrementing the `size` and changing the `EnrolledIn` predicate so that it is true for `st, crs`.

Unfortunately, the specification of *enrolInCourse* is subject to at least two readings. Moreover, the more intuitive reading is the harder one to formulate in Predicate Logic. To see the problem, consider two implementations of this specification. The first results in exactly two changes to a program state, as required by the postcondition. The second makes these changes, but also changes some other predicate, such as `CourseCompleted`, or `EnrolledIn` at some other argument. The more intuitive, "tighter", reading of the specification is inconsistent with the second implementation because that implementation does more than what is necessary to make the postcondition true. In other words, the intuitive reading includes a clause of the form "...and nothing else changes" to circumscribe the modifications effected by a procedure.

The second, "looser", reading of the specification does not include the "...nothing else changes" clause and is consistent with both implementations mentioned earlier. This reading is less intuitive in that it merely places a lower bound on the effects of the procedure that has been specified. On the other hand, this reading has two advantages: It is less restrictive, thus offering greater implementation freedom -- a desirable property of specifications. And it affords a simple formalization in Predicate Logic: first, view states as models for formulae involving unprimed predicates and functions -- the "variables" that the program will update; the eventual implementation must then establish a state where the postcondition formula evaluates to true, with primed predicates evaluated in the final state, and unprimed ones in the initial state. Second, note that the priming of predicates allows us to put together the theories of the initial state and final state into a single theory for the entire pre/post-condition formula. Note that the relationship between predicates `R` and `R'` is entirely through convention (and through the postcondition) -- they may as well be called by completely different names. In particular, any proof obligation of the form mentioned in the introduction which involves a program state invariant `I` and a procedure `p` with pre/postconditions `pre-p(`$\underline{x}$`)`, `post-p(`$\underline{x}$`)` has to be stated as

---

[4] In general, function symbols used in formulas will begin with a lower case letter and predicate symbols with upper case ones, while class names will be fully in capital letters.

[5] Preconditions are assumed to define partial functions, so that if somehow the procedure is invoked with the precondition false, it will not terminate. Normally, additional specifications would be conjoined to this one to specify exception handling.

$$\forall \underline{x}[I \wedge \text{pre-p}(\underline{x}) \wedge \text{post-p}(\underline{x}) \Rightarrow \textbf{\textit{primed}}(I)]$$

where ***primed***(I) represents the invariant I with all its predicates and functions primed. Now suppose that we have some simple state invariant $\forall y.Q(y)$, where Q is a predicate; then the proof obligation amounts to demonstrating the following formula:

```
∀ st,crs [
    ∀x.Q(x) ∧
     (size(crs) < classLimit(crs) ∧  ¬EnrolledIn(st,crs)) ∧
         (size'(crs) = size(crs) + 1 ∧ EnrolledIn'(st,crs))
     ⇒ ∀x.Q'(x) ]
```

This cannot be proven because the postcondition of the procedure says nothing about the predicate Q'. Therefore, such proof obligations cannot be achieved using the looser reading of the specification unless the specifier takes the trouble to mention *explicitly* in her specification not just the things that are changed by the procedure, but also all those that are not. Thus if the specifier wants to state that "...nothing else changes", she has to state so explicitly in the specification by adding a clause of the form

$$\forall \underline{x}[(P(\underline{x}) \Rightarrow P'(\underline{x})) \wedge (\neg P(\underline{x}) \Rightarrow \neg P'(\underline{x}))]$$

or, equivalently,

$$\forall \underline{x}[P(\underline{x}) \equiv P'(\underline{x})]$$

for every predicate P other than EnrolledIn (such as Q), and a clause of the form

$$\forall \underline{x}[f(\underline{x}) = f'(\underline{x})]$$

for every function f other than size. In addition, there have to be clauses that state that EnrolledIn and size remain unchanged for all arguments other than st, crs:

```
∀x [ x ≠ crs  ⇒  size(x) = size'(x)]

∀x, y [ x ≠ st ∨ y ≠ crs  ⇒
      (EnrolledIn(x, y) ≡ EnrolledIn'(x, y))]
```

Assuming that program states are described by only two functions, size and classLimit, and two predicates, EnrolledIn and Q, a complete specification of enrolInCourse which states explicitly that "...nothing else changes" is as follows:

enrolInCourse(st,crs)

```
PRE: size(crs) < classLimit(crs)  ∧  ¬EnrolledIn(st,crs)

POST: size'(crs) = size(crs) + 1  ∧  EnrolledIn'(st,crs) ∧
    ∀x [ x ≠ crs  ⇒
          size(x) = size'(x)] ∧
    ∀x, y [ x ≠ st ∨ y ≠ crs ⇒
          (EnrolledIn(x, y) ≡ EnrolledIn'(x, y))] ∧
    ∀x [classLimit(x) = classLimit'(x)] ∧
    ∀x [Q(x) ≡ Q'(x)]
```

The extra clauses that were needed to state explicitly that "...nothing else changes" have been called *frame axioms* [McCarthy69] while the general problem of stating succinctly the "...nothing else changes" clause is precisely the *frame problem*..

We hope that the reader already agrees that requiring the specifier to provide the frame axioms makes specification a lengthier, less perspicuous and more error-prone process. Failure to deal with the frame axioms, on the other hand, limits the extent to which formal properties of specifications can be proven.
In the rest of this section we focus on the ways such problems have been treated in existing specification languages, and on presenting three special kinds of specifications, for which these solutions (as well as the standard FOPC approach) appear to be inadequate.

2.2 Stating Frame Axioms in Specification Languages

In practice, specification languages are enhanced to allow the specifier to state explicitly, but more conveniently, what elements of a program state remain unchanged. To see such features in action, we consider several widely known specification languages such as Z [Spivey89] and VDM [Jones86]. These are based on the notation of mathematics and set theory, where the functions and predicates of our predicate-calculus notation are naturally converted to functions and relations, which are often viewed as sets of tuples. In such a case, the predicate assertion `EnrolledIn'(st,crs)` translates naturally to `(st,crs) ∈ enrolledIn'`. The notation of set-theory is then normally used to express some part of the frame axioms by stating

$$\mathtt{enrolledIn'} = \mathtt{enrolledIn} \cup \{(\mathtt{st},\mathtt{crs})\}$$

and

$$\mathtt{size'} = \mathtt{size} \oplus \{\ \mathtt{crs} \mapsto \mathtt{size(crs)+1}\ \}$$

These statements have the effect of asserting that `EnrolledIn` and `size` do not change at arguments other than `st` and `crs`. We are then left only with the task of stating that enroling in a course does not modify the `classLimit` or other predicates such as `Q`. For this task, Z provides the Ξ predicate constructor, which allows one to take a named list of variables $N = (x_1, x_2, ..., x_k)$, and produce the assertion

$$\Xi N =_{def} x_1' = x_1 \wedge x_2' = x_2 \wedge ... \wedge x_k' = x_k$$

By naming the list of remaining variables, `Rest = (classLimit,Q)`, we can then express the desired meaning of `enrol` as the Z schema[6]

enrolInCourse

```
st : STUDENT
crs : COURSE
Ξ Rest
-------------------------------------------------------------
size(crs) < classLimit(crs)   ∧   (st,crs) ∉ enrolledIn ∧
 enrolledIn' = enrolledIn ∪ {(st,crs)} ∧
 size' = size ⊕ {crs ↦ size(crs) + 1}
```

This approach works well if we can modularize the presentation to give names to appropriate subsets of variables, such as `Rest` above.

Many other specification languages, such as Larch [Guttag85] and VDM [Jones86], take a second approach, whereby a procedure specifier is required to identify the variables that it *might* modify, with the *implicit* assertion that any variable not so declared remains unchanged. (This implicit assertion must then be expanded by some automatic background tool or used by special-purpose theorem provers.) Specifically, in VDM the specification of `enrol` would indicate that it needs access only to external variables `size`, `classLimit` and `enrolledIn`, with the second one being a "read only", so that it might modify at

---

[6] Both here and elsewhere, we have chosen to use a sometimes more self-explanatory syntax in order to avoid frequent detours into notation. For example, according to proper Z style, the input parameters should be named `st?` and `crs?`, and we should also have included statements of the form Δ `enrolledIn` in order to introduce the symbols `enrolledIn` and `enrolledIn'`.

most `size` and `enrolledIn`. (For variety, in this case we will model class enrolments as a function from students to sets of courses. Also, we will take the liberty of continuing to use theconvention of priming the variables in the post-state, although languages such as VDM and COLD use different syntactic conventions.)

```
ENROL-IN-COURSE(st : STUDENT, crs : COURSE)
ext
          wr size        :  CourseToInt
          rd classLimit  :  CourseToInt
          wr enrolledIn  :  Student → Course-set
pre       size(crs) < classLimit(crs) ∧
            crs ∉ enrolledIn(st)
post      enrolledIn' = enrolledIn  ∪ {(st,crs)}   ∧
            size' = size + { crs  ↦ 1+size(crs) }
```

The specification language COLD [Jonkers91] has an even more refined way of stating so-called "modification rights", by using complex expressions that delimit the subset of the state space over which the operation can act differently than the identity mapping. For example, in the COLD specification below, the line `MOD` indicates that only the enrolment of the specific student `st` , and the size of the specific course `crs` may change.

```
PROC enrolInCourse: STUDENT # COURSE →
IN   st,crs
PRE  size(crs) < classLimit(crs) ∧
      (crs) ∉ enrolledIn(st)
MOD  enrolledIn(st), size(crs)
POST enrolledIn'(st) = enrolledIn(st) ∪ { crs }   ∧
       size'(crs) = size(crs) +1
```

Larch [Guttag85] provides a "*modifies at most*" clause with similar capabilities.

Clearly, these techniques work quite well in this simple case. However, we present next a series of examples that do not fare as well.

2.3 Conditional Specifications

Suppose we now want to redefine *`enrolInCourse`* so that if there is no room left in the course but the student is in her fourth year of school, her request is added to a waiting list; otherwise, the student is sent a notice rejecting her request. Moreover, assume that `waitSize` and `rejectSize` are functions counting respectively the number of enrollment attempts on a waiting list or rejected. Returning to our Predicate Calculus notation, we might be inclined to state the new postcondition of *`enrolInCourse`* as

```
size(crs) < classLimit(crs) ⇒
    (size'(crs) = size(crs) + 1 ∧ EnrolledIn'(st,crs))        ∧
size(crs) ≥ classLimit(crs) ∧ Year(st, 4) ⇒
    (waitSize'(crs) = waitSize(crs) + 1 ∧ Waiting'(st,crs))    ∧
size(crs) ≥ classLimit(crs) ∧ ¬Year(st, 4) ⇒
    (rejectSize'(crs) = rejectSize(crs) + 1 ∧ Rejected'(st,crs))
```

But once again, we run into the frame problem: we forgot to state for each branch of the conditional what portions of the program state remain unchanged. We must therefore write

```
size(crs) < classLimit(crs) ⇒
    (size'(crs) = size(crs) + 1 ∧ EnrolledIn'(st,crs)) ∧
        (∀x [ x ≠ crs  ⇒ size(x) = size'(x)] ∧
```

```
            ∀x, y [ x ≠ st ∨ y ≠ crs  ⇒
                  (EnrolledIn(x, y) ≡ EnrolledIn'(x, y))] ∧
            ∀x [ waitSize(x) = waitSize'(x)] ∧
            ∀x, y [Waiting(x, y) ≡ Waiting'(x, y)] ∧
            ∀x [ rejectSize(x) = rejectSize'(x)] ∧
            ∀x, y [Rejected(x, y) ≡  Rejected'(x, y)] ) ∧
  size(crs) ≥ classLimit(crs) ∧ Year(st, 4) ⇒
      (waitSize'(crs) = waitSize(crs) + 1 ∧
                                     Waiting'(st,crs)) ∧
          ( ∀x [ x ≠ crs   ⇒ waitSize(x) = waitSize'(x)] ∧
            ∀x, y [ x ≠ st ∨ y ≠ crs  ⇒
                  (Waiting(x, y) ≡ Waiting'(x, y))] ∧
            ∀x[size(x) = size'(x)] ∧
            ∀x, y[EnrolledIn(x, y) ≡ EnrolledIn'(x, y)] ∧
            ∀x[rejectSize(x) = rejectSize'(x)] ∧
            ∀x, y[Rejected(x, y) ≡  Rejected '(x, y)] ) ∧
  size(crs) ≥ classLimit(crs) ∧ ¬Year(st, 4) ⇒
      (rejectSize'(crs) = rejectSize(crs) + 1 ∧
                                     Rejected'(st,crs)) ∧
          ( ∀x [ x ≠ crs ⇒ rejectSize(x) = rejectSize'(x)] ∧
            ∀x, y[ x ≠ st ∨ y ≠ crs  ⇒
                  (Rejected(x, y) ≡ Rejected'(x, y))] ∧
            ∀x [size(x) = size'(x)] ∧
            ∀x, y[EnrolledIn(x, y) ≡ EnrolledIn'(x, y)] ∧
            ∀x [waitSize(x) = waitSize'(x)] ∧
            ∀x, y[Waiting(x, y) ≡  Waiting'(x, y)] ) ∧  ...
```

To complete this postcondition, we must also add frame axioms for functions and predicates that are not affected in any of the cases by the procedure (the function `classLimit`, for example). Specification of the frame axioms here is even more cumbersome, partly because each case requires a different set of such axioms.

Even in Z , VDM, or COLD, we would need to say at the beginning that `rejected`, `rejectSize`, `waiting`, `waitSize`, `size`, and `enrolledIn` all might be modified, and then state for each branch what does/doesn't change, as in

```
  size(crs) < classLimit(crs) ⇒
     [enrolledIn' = enrolledIn  ∪ {(st,crs)} ∧
      size’ = size ⊕ { crs |→  size(crs) + 1}
      waitSize’=waitSize   ∧ waiting' = waiting ∧
      rejectSize’=rejectSize  ∧ rejected' = rejected]  ∧
  size(crs) ≥ classLimit(crs) ∧ year(st)= 4  ⇒
      [waiting' = waiting  ∪ {(st,crs)} ∧
      waitSize’ = waitSize ⊕ { crs |→ waitSize(crs) + 1}
      size’= size  ∧  enrolledIn' = enrolledIn ∧
      rejectSize’= rejectSize  ∧ rejected' = rejected ]  ∧
  size(crs) ≥ classLimit(crs) ∧ ¬year(st)=4 ⇒
      [rejected' = rejected  ∪ {(st,crs)} ∧
      rejectSize’ = rejectSize ⊕ { crs |→  waitSize(crs) + 1}
      size’= size  ∧  enrolledIn' = enrolledIn ∧
      waitSize’= waitSize  ∧ waiting' = waiting ]
```

The problem here is that we have to state *ahead of time* the list of things that *might* change in the procedure, and this forces us to state explicitly for any branch of the conditional the things that do not change.[7]

2.4 Conjoining Specifications

Consider now a procedure `switchCourse` which involves dropping one course and enrolling in another. Suppose that the system being specified already includes the specification of procedures `enrolInCourse` and `dropCourse`, and we are now asked to specify a procedure `switchCourse`. It is natural, and in fact desirable from the point of view of reuse and propagation of changes, to try to define `switchCourse` by making use of the other two specifications. In Z this is accomplished simply by using the schema composition notation [Spivey89]:

$$\mathtt{switchCourse(st,\ crs_1,\ crs_2)} =_{\mathrm{def}}$$
$$\mathtt{dropCourse(st,crs_1)}\ \|\ \mathtt{enrolInCourse(st,crs_2)}$$

where "‖" is a binary operation on specifications which conjoins its operands (in our case conjoining their pre- and postconditions[8]). Unfortunately, such an operation leads to meaningless specifications if the two conjoined procedures include their respective frame axioms. For example, assume that *dropCourse* is defined as follows:

dropCourse

```
st : STUDENT
crs : COURSE
Ξ Rest
--------------------------------------------
(st,crs) ∈ enrolledIn ∧
 enrolledIn' = enrolledIn − {(st,crs)} ∧
 size' = size ⊕ {crs ↦ size(crs) − 1}
```

Then, clearly, the conjunction of the postconditions of *enrolInCourse* and *dropCourse* is inconsistent since it includes, among other clauses,

$$\mathtt{enrolledIn'} = \mathtt{enrolledIn} - \{\mathtt{(st,crs_1)}\} \wedge$$
$$\mathtt{enrolledIn'} = \mathtt{enrolledIn} \cup \{\mathtt{(st,crs_2)}\}$$

Note that exactly the same situation occurs in VDM and even COLD: the conjunction of the post-conditions will have the form

```
POST   enrolledIn'(st) = enrolledIn(st) ∪ { crs2 } ∧
       size'(crs2) = size(crs2) +1
       enrolledIn'(st) = enrolledIn(st) − { crs1 } ∧
       size'(crs1) = size(crs1) −1
```

which is inconsistent for `enrolledIn'(st)`. The problem in this case is caused by the use of set-theoretic notation to state part of the frame axiom. This example suggests that the "...nothing else changes" clause should read, more precisely, "...nothing else changes, <u>unless otherwise stated</u>". Note that if we had

---

[7] It has been suggested to us (James Power, personal communication) that Z's facility for *hiding* identifiers might be used to abbreviate the above to some extent: if ΞV is a schema asserting that all variables remain unchanged , then to each case of the conditional one can conjoin ΞV with the variables modified in that case being hidden.

[8] Note that such a conjoin operation makes no assumptions on the order in which the two conjoined procedures will be carried out --- it only ensures that the effect of both is carried out.

been able to describe *enrolInCourse* and *dropCourse* with post-conditions of the form `crs ∈ enrolledIn'(st)` and `crs ∉ enrolledIn'(st)`, then the final conjunction would have had the consistent form

$$\texttt{crs}_1 \notin \texttt{enrolledIn'(st)} \wedge \texttt{crs}_2 \in \texttt{enrolledIn'(st)}$$

However, this requires different frame axioms for `EnrolledIn` in the context of *enrolInCourse*, *dropCourse* and *switchCourse*. If frame axioms are treated as textual elements of a procedure specification, language facilities such as the schema conjunction operation may be rendered problematic or at best ineffective.

2.5 Inheritance and Object-Oriented Specifications

The above problems could be shrugged off as mere annoyances. Unfortunately, they lie at the heart of object-oriented specifications, which appear to be of increasing interest (see, for example, [Schuman87], [Duke90], [Alencar94], [Marshall91], and the collections [Stepney92], [Lanno94]). Two distinguishing features of object-oriented programming languages (OOPLs, for short) are taken to be (i) the presence of class hierarchies with inheritance, where what is stated in a class need not be repeated for its subclasses; (ii) the selection of procedure versions (*methods*, in OOPL jargon) to execute based on the type of the arguments. It is exactly these features that distinguish object-oriented languages such as Eiffel [Meyer88] or Taxis [Mylopoulos80], from those that merely support data encapsulation, such as ADA or CLU. The arguments for the benefits of inheritance presented for OOPLs also apply to object-oriented specification languages (e.g., [Borgida84]) and include abbreviating specifications, propagating changes and encouraging reuse.

As argued, among others, in [Meyer88,Borgida84], an important and desirable property of subclasses with inheritance is the ability to *specialize* the specification of superclasses. For example, if the class `EDUCATOR` has attribute `degrees`, which is required to be a subset of the enumeration `{BA,BS,Ms,Phd}`, then it is desirable to allow the type of degree to be specialized to `{PhD}` for the subclass `PROFESSOR` of `EDUCATOR`. The same principle is applicable to the specialization of methods [Meyer88, Borgida81]. For example, if a class of equations has a method `solve` that finds roots within an error bound of 0.001 say, then some subclass (for which a special technique might be found, or which is particularly important) should be able to offer the same method but with a tighter error bound, of 0.000001 say.

As it turns out, inheritance is accomplished in almost all object-oriented specification languages by simply conjoining the specification of the superclass(es) to the additional assertions associated with the subclass. For example, [Alencar 94] shows how in OOZE one can obtain the class `SUPER_PREMIUM_ACCOUNT` from the simple conjunction of two parent classes: `CHEQUE_BOOK_ACCOUNT` and `PREMIUM_ACCOUNT`. However, as we have seen in Section 2.4, conjunction of specifications will run into serious trouble unless we take the loosest possible way of stating post-conditions. In particular, in any specification language supporting "bulk" data-types such as sets, relations, records, etc., if a procedure `P` specifies a change to a variables of such a type, then no specialization of `P` can ever modify any other aspect of `s`, if some part of the frame assertion appears explicitly in the post-condition of `P`. For example, if `P` calls for the removal from set `s` of all values satisfying some predicate Ψ, then no specialization of `P` may strengthen this to some stronger predicate Φ, which removes additional elements from `s`. Or if `s` records some property of individuals, such as salary or size, then changing `s(a)` for one object prevents `s(b)` from being modified for any other object `b` in a specialization[9]. Such limitations are particularly bothersome in object-centered information systems that rely on data bases (e.g., [Schewe91]), where data about individuals is globally visible (encapsulation is not the issue here), and is recorded by making attributes be binary relations/functions to sets, which are bulk data types of the kinds subject to the above mentioned restrictions.

We illustrate this problem with several further examples. Consider a university where students in courses are eventually assigned (absolute) numeric grades from 0 to 100, and we want to associate with every course a list of honors students in it. Using a composite of notations, we could model this as the class:

---

[9] The very specific framing statements of COLD and Larch can actually deal with this problem.

```
class COURSE
 enrolment:  Set of   STUDENT
 grade:      STUDENT  ⇸   0..100
 honors:     Set of  STUDENT   initially empty
...
methods:

findHonors()
    PRE:  domain grade = enrolment
    MOD:  honors
    POST: honors' = { s ∈ enrolment |  grade(s) > 85}

...
```

Suppose now that for senior courses, among others, we wish to lower the requirement for honors to 80%, since grades are presumably harder to get. The obvious specification of this would be something like

```
class SENIOR-COURSE  subclass of COURSE
...
methods:

findHonors()
    POST: honors' = { s ∈ enrolment | grade(s) > 80}
...
```

This would result in the specification of `SENIOR-COURSE::`*`findHonors`*`()` having post-condition

$$\text{honors}'=\{ s \mid \text{grade}(s) > 85\} \;\wedge\; \text{honors}'=\{ s \mid \text{grade}(s) > 80\}$$

which, at best, would imply that there are no students with grades between 80 and 85. Notice that the specialization we have in mind is conceptually consistent: the students assigned honors in the original courses would still get them, but additional ones would also do so. (This is in contrast with "non-strict" inheritance, as practiced in Smalltalk, say, where the specialized procedure may do *anything,* even contradictory to its more general version.) The problem here lies in the frame assertion implicit in the set-theoretic notation -- the `MOD` clause already restricts as carefully as possible the state variables that will change; but had we stated the post-conditions in the form,

$$\forall x/\text{enrolment} \,.\, \text{grades}(x)>85 \;\Rightarrow\; x \in \text{honors}'$$

the loose reading would not prevent `honors` from containing all students, for example.

As another example, consider the class `COURSE-WITH-COREQ`, a subclass of `COURSE` where we wish to maintain a list of courses which need to be taken at the same time as this course. In this case, the `drop` method must be refined for `COURSE-WITH-COREQ` to remove the student who is droping this course from the enrolments of all corequisites. Here are partial specifications of the two classes, using the notation of Predicate Calculus in this case, together with some of the explicit frame axioms:[10]

```
class COURSE
HasEnrolled:  COURSE x STUDENT
methods:
 drop(self: COURSE, st: STUDENT)
      PRE:   HasEnrolled(self,st)
      POST: ¬HasEnrolled'(self,st)
           ∧∀x/STUDENT,y/COURSE[ x ≠ st ∧ y ≠ self  ⇒
```

[10] In this case, unlike standard object-oriented specifications, we introduce the variable `self` explicitly, rather than having for each method a privileged, but unnamed, first argument.

```
                (HasEnrolled(y,x) ≡ HasEnrolled'(y,x))]
```

<u>class</u> COURSE-WITH-COREQ <u>subclass of</u> COURSE

```
Coreqs: COURSE-WITH-COREQ x COURSE
...

  drop(self: COURSE, st:STUDENT)
      POST: ∀x/COURSE . Coreq(self,x) ⇒ ¬HasEnrolled(x,st)]
                                       ∧ ... (other frame axioms)
```

By inheritance, the full specification of `COURSE-WITH-COREQ::`*drop* is obtained by conjoining the assertions stated on `COURSE::`*drop* with the additional material associated with `drop` in `COURSE-WITH-COREQ`. However, this is clearly inconsistent because `COURSE::`*drop* states that `HasEnrolled` only changes at one argument, while `COURSE-WITH-COREQ::`*drop* calls for additional changes whenever there are corequisites.

It is important to note that it is our intention that in any system implementation generated from this specification, an object could be made an instance of the class `COURSE` without necessarily being an instance of `COURSE-WITH-COREQ`. Therefore we cannot leave out the frame axioms from the definition of `COURSE::`*drop* , or if we do so, we need to rename `COURSE` to `COURSE*` -- a dummy class introduced strictly for technical reasons --- and make `COURSE` its subclass with the frame assertion added to drop. We find the latter alternative less acceptable from the point of view of notational convenience and naturalness.

As a final example, suppose we track, among others, the addresses of employees, and wish to provide a method, *move*, for changing the address of an employee:

<u>class</u> EMPLOYEE

```
livesAt: EMPLOYEE ⇸   ADDRESS
...
```

EMPLOYEE.*move* (self: EMPLOYEE, new:ADDRESS)

```
    MOD:  livesAt
    POST: livesAt'(self)=new
            ∧  ∀y.(y≠self ⇒ livesAt(y)=livesAt'(y))
            ∧ ... (other frame axioms)
```

Now, for the subclass of married employees, we need to change *move* so that the spouse's address is also changed:

MARRIED_EMPLOYEE.*move*

```
    POST: livesAt'(spouse(self))=new
           ∧ ∀y.(y≠self ∧ y≠spouse(self)) ⇒ livesAt(y)=livesAt'(y)
           ∧ ... (other frame axioms)
```

Only a very refined "modifies at most" facility has a hope of dealing with this situation, if `EMPLOYEE::`*move* has <u>MOD</u>: `livestAt(self)` while `MARRIED-EMPLOYEE.`*move* has <u>MOD:</u> `livesAt(spouse(self))`.

2.6 Conclusions from the Examples

The necessity of stating explicitly that some parts of a program state remain unchanged has been shown to make specifications longer, more difficult to comprehend and change, more error prone (what if the specifier overlooks some frame axioms?) and presents an obstacle to aspects of inheritance, and hence object-orientation. In fact, without extending the language features, object-oriented formal specifications seem possible only if one assumes that local variables of a class are not visible to its subclasses, and procedure specifications cannot be refined in subclasses --- a restriction not shared by popular object-oriented programming languages such as C++, Smalltalk and Eiffel.

Even if one were prepared to accept the above, there is an additional cost that arises from the fact that for a specification involving n procedures and m functions and predicates describing program states, there are going to be in general $O(\mathtt{m*n})$ frame axioms. These can make proofs of consistency of a specification prohibitively expensive, thereby removing a major argument in favor of formal program specifications.

3. A Novel Approach to the Frame Problem in Procedure Specifications

We propose a new approach to the frame problem in procedure specifications inspired by work reported in [Reiter91] on the frame problem in AI planning.

3.1 Foundations of the Approach

In short, our proposed approach adds a second component to the specification process: in addition to specifying post-conditions, we also look for a set of assertions that explain the circumstances under which each predicate or function might change value from one program state to the next. There are several ways in which this can be achieved, and for present purposes we have chosen a moderately complex approach, which could in fact be used to reason more generally about procedures. The approach reifies procedures by adding *actions* as a new sort in the specification language, with all the procedure names being function symbols of this sort, and taking as arguments the formal parameters of the procedure. For example, the procedure *enrolInCourse* would be formally named by the two-place function symbol `enrolInCourse`, so that enrolling Ann into CS100 would be represented by the term `enrolInCourse(ann,cs100)`. To talk about actions, we also introduce the single variable α, and the predicate `Occur()`, which takes an action argument. The intended interpretation of `Occur()` is that the procedure which is its argument executed successfully.[11]

For every procedure $p(\underline{x})$ with specification $(\mathrm{pre\text{-}p}(\underline{x}),\ \mathrm{post\text{-}p}(\underline{x}))$ we generate first a so-called *effect axiom*

$$\forall \underline{x}[\mathrm{Occur}(\mathrm{p}(\underline{x})) \Rightarrow \mathrm{pre\text{-}p}(\underline{x}) \wedge \mathrm{post\text{-}p}(\underline{x})]$$

describing the effect of the procedure; i.e., if the procedure was executed (and this is often an external decision, like choosing to enroll in a course), this axiom assures us of some necessary conditions that must have held. For example, the effect axiom for the initial specification of *enrolInCourse* would be

$$\begin{aligned}&\forall x, y[\ \mathrm{Occur}(\mathrm{enrolInCourse}(x, y)) \Rightarrow\\ &\qquad (\mathrm{size}(y) < \mathrm{classLimit}(y) \wedge \neg\ \mathrm{EnrolledIn}(x, y)) \wedge\\ &\qquad (\mathrm{size}'(y) = \mathrm{size}(y) + 1 \wedge\ \mathrm{EnrolledIn}'(x, y))]\end{aligned}$$

Note that this omits frame axioms; these will be specified separately and from a different perspective, through *explanation closure axioms* (called *change axioms* for short). For every predicate `R`, the specifier must write two such axioms describing under what circumstances `R` might change truth value (from positive to negative or negative to positive); the circumstances include the actions that took place, their parameters, the values where `R` changed as a result, and complex conditions involving both.

$$\begin{aligned}&\forall\alpha\forall \underline{y}[\ \mathrm{R}(\underline{y})\ \wedge\ \neg\mathrm{R}'(\underline{y})\ \wedge\ \mathrm{Occur}(\alpha) \Rightarrow\\ &\qquad \exists \underline{z}_1(\alpha\ =\ \mathrm{p}_1(\underline{z}_1)\ \wedge \Psi^{\mathrm{R},\mathrm{p}_1}(\underline{w}_1))\ \vee\\ &\qquad \exists \underline{z}_2(\alpha\ =\ \mathrm{p}_2(\underline{z}_2)\ \wedge \Psi^{\mathrm{R},\mathrm{p}_2}(\underline{w}_2))\ \vee\end{aligned}$$

---

[11] We leave to a different paper the matter of termination, and the possibility of several actions occuring at the same time.

$$\ldots$$

$$\forall\alpha\forall\underline{y}[\ \neg R(\underline{y})\ \wedge\ R'(\underline{y})\ \wedge\ Occur(\alpha) \Rightarrow$$
$$\exists\underline{z}_1(\alpha = q_1(\underline{z}_1)\ \wedge \Psi^{R,q_1}(\underline{w}_1))\ \vee$$
$$\exists\underline{z}_2(\alpha = q_2(\underline{z}_2)\ \wedge \Psi^{R,q_2}(\underline{w}_2))\ \vee$$
$$\ldots$$

Here p_i, q_i are action symbols and Ψ^{R,p_i} (resp. Ψ^{R,q_i}) describes the circumstances under which p_i (resp. q_i) can be executed to change R, and $\underline{w}_j$ is a vector of values chosen from among the constants and variables in $\underline{y}$ and $\underline{z}_j$.

For functions, a single change axiom is sufficient:

$$\forall\alpha\forall\underline{y}[\ f(\underline{y})\ \neq f'(\underline{y})\ \wedge\ Occur(\alpha) \Rightarrow$$
$$\exists\underline{z}_1(\alpha = p_1(\underline{z}_1)\ \wedge \Psi^{f,p_1}(\underline{y}_1))\ \vee$$
$$\exists\underline{z}_2(\alpha = p_2(\underline{z}_2)\ \wedge \Psi^{f,p_2}(\underline{y}_2))\ \vee$$
$$\ldots$$

These axioms offer a *state-oriented* rather than a *procedure-oriented* perspective to the frame problem. Thus, instead of asking the specifier to assert what each procedure *does not change*, we ask her to declare what procedures *could have effected a change* to a particular element of the program state.

As an example of change axioms, consider the predicate EnrolledIn, and suppose that the only procedure affecting it is the procedure *enrolInCourse*, though there may be many other procedures around, such as *assignRoom*, *assignTeacher*, etc. The change axioms then are

$$\forall\alpha\ \forall s, c$$
$$(EnrolledIn(s,c)\ \wedge \neg EnrolledIn'(s,c) \wedge Occur(\alpha)\ )\ \ \Rightarrow false$$

indicating that no procedure removes students from courses, and

$$\forall\alpha\,\forall st, crs$$
$$(\neg EnrolledIn(st,crs)\ \wedge EnrolledIn'(st,crs) \wedge Occur(\alpha)\ ) \Rightarrow$$
$$(\alpha = enrolInCourse(st,crs))]$$

indicating that only *enrolInCourse* can add a student to the course. Likewise, the change axiom for the function size is

$$\forall\alpha\forall crs\ .\ [size(crs) \neq size'(crs) \wedge Occur(\alpha)] \Rightarrow$$
$$\exists st\ .\ \alpha = enrolInCourse(st,crs)$$

Observe how elegantly these axioms capture the fact that in a closed system (where all procedures are known) the only source for change of the predicate EnrolledIn is the procedure *enrolInCourse*. This means that the pre/postcondition specifications of all the other procedures need not say anything about how they affect EnrolledIn. Also note that the change axiom for size need not explain the nature of the change to that function. This information is already captured by the effect axioms of the procedures.

Incidentally, proof obligations concerning invariants maintained by a procedure p(x) could now be recast to simply assume as a premise the occurrence of a procedure:

$$\forall\underline{x}[I \wedge Occur(p(\underline{x})) \Rightarrow \textbf{\textit{primed}}(I)]$$

together with some assurances that no other action has occured.

We emphasize again that here it is the responsibility of the specifier to write down the change axioms, and that these may need to be modified as new procedures are added or old ones are altered. For example, if we now were to add a procedure *dropCourse*, with its own specification

```
dropCourse(st,crs)
PRE: EnrolledIn(st,crs)

POST: size'(crs) = size(crs)-1  ∧  ¬EnrolledIn'(st,crs)
```

then we would have to add appropriate clauses to the change axioms of `EnrolledIn` and `size`. This can be done quite cleanly by adding disjuncts:

```
∀α∀st, crs .
EnrolledIn(st,crs) ∧ ¬EnrolledIn'(st,crs) ∧ Occur(α) ⇒
             false ∨ (α = dropCourse(st,crs))

∀α∀crs .
size(crs) ≠ size'(crs) ∧ Occur(α) ⇒
     [∃st.α = enrolInCourse(st,crs) ∨
      ∃st.α = dropCourse(st,crs)]
```

It is worth noting that although the frame axioms about a single procedure are now spread over possibly several change axioms, the process of thinking about these axioms is systematic. In particular, if there is a change in the postcondition of some procedure `p`, then we must only consider the change axioms for those predicates which were added, dropped or modified by the change; and even here, we only need to consider that sub-formula of the explanation axiom which starts with $\exists \underline{z}(\alpha = \mathtt{p}...)$. Furthermore, any time the state is extended with another predicate or function, there is no concern that this now might be inadvertently be left open to change by some procedure specification.

In order to reason appropriately with the above axioms, we will need to add two additional axioms dealing with the reification of procedures: one stating that the only one action occurs at one time

$$\exists!\alpha.\mathtt{Occur}(\alpha)$$

and another one that the actions are distinct iff their names and arguments are distinct (the so called *unique name* axioms): for $i \neq j$,

$$\forall \underline{x},\underline{y}[(p_i(\underline{x}) \neq p_j(\underline{y})) \wedge (p_i(\underline{x})=p_i(\underline{y}) \Rightarrow \underline{x}=\underline{y})]$$

Details of the axiomatization and a discussion of its limitations can be found in [Reiter95]. In particular, as described here, change axioms do not always have their intended effects, for instance in cases involving non-deterministic procedures. For an approach to change axioms in the presence of non-determinism, see [Reiter95].

4.2 Frame Problem Variations Revisited

To show how this approach handles the problems raised earlier, consider again the examples of sections 2.3-2.5.

With the frame axioms out of the way, the specification of the more complex enrolment's post-condition in Section 2.3 would be as at the beginning of that section:

```
size(crs) < classLimit(crs) ⇒
    (size'(crs) = size(crs) + 1 ∧ EnrolledIn'(st,crs))        ∧
size(crs) ≥ classLimit(crs) ∧ Year(st, 4) ⇒
```

$$\begin{array}{l} \quad (\mathtt{waitSize'(crs)} = \mathtt{waitSize(crs)} + 1 \wedge \mathtt{Waiting'(st,crs)}) \quad \wedge \\ \mathtt{size(crs)} \geq \mathtt{classLimit(crs)} \wedge \neg\mathtt{Year(st,\ 4)} \Rightarrow \\ \quad (\mathtt{rejectSize'(crs)} = \mathtt{rejectSize(crs)} + 1 \wedge \mathtt{Rejected'(st,crs)}) \end{array}$$

The change axioms for EnrolledIn and size would be:

$$\begin{array}{l} \forall\alpha\,\forall \mathtt{st,\ crs} \\ \quad \neg\mathtt{EnrolledIn(st,crs)} \wedge \mathtt{EnrolledIn'(st,crs)} \wedge \mathtt{Occur}(\alpha) \Rightarrow \\ \qquad (\alpha = \mathtt{enrolInCourse(st,crs)} \wedge \mathtt{size(crs)} < \mathtt{classLimit(crs)}) \end{array}$$

$$\begin{array}{l} \forall\alpha\forall \mathtt{st,\ crs} \\ \quad \mathtt{EnrolledIn(st,crs)} \wedge \neg\mathtt{EnrolledIn'(st,crs)} \wedge \mathtt{Occur}(\alpha) \Rightarrow \\ \qquad \mathtt{false} \end{array}$$

$$\begin{array}{l} \forall\alpha\,\forall \mathtt{crs} \\ \quad \mathtt{size(crs)} \neq \mathtt{size'(crs)} \wedge \mathtt{Occur}(\alpha) \quad \Rightarrow \\ \qquad \exists \mathtt{st}\,(\alpha = \mathtt{enrolInCourse(st,crs)} \wedge \mathtt{size(crs)} < \mathtt{classLimit(crs)}) \end{array}$$

We would have similar change axioms for waiting, sizeWaiting, etc.

Conjoining of procedure specifications also works out without surprises, as long as we state the post-conditions in a "loose" way. The specification of *enrolInCourse*, *dropCourse* and *switchCourse* in Z might then be

$$\begin{array}{l} \mathtt{enrolInCourse} =_{\mathtt{def}} [\mathtt{st:Student,\ crs:Course,}\ \Delta\,\mathtt{enrolledIn,size}\ | \\ \qquad \mathtt{size(crs)} < \mathtt{classLimit(crs)} \wedge \\ \qquad \mathtt{(st,crs)} \in \mathtt{enrolledIn'} \wedge \mathtt{size'(crs)} = \mathtt{size(crs)} + 1] \end{array}$$

$$\begin{array}{l} \mathtt{dropCourse} =_{\mathtt{def}} [\mathtt{st:Student,\ crs:\ COURSE,}\ \Delta\,\mathtt{enrolledIn,size}\ | \\ \qquad \mathtt{(st,crs)} \in \mathtt{enrolledIn} \wedge \\ \qquad \mathtt{(st,crs)} \notin \mathtt{enrolledIn'} \wedge \mathtt{size'(crs)} = \mathtt{size(crs)} - 1] \end{array}$$

$$\begin{array}{l} \mathtt{switchCourse(st,\ crs_1,\ crs_2)} =_{\mathtt{def}} \\ \quad \mathtt{dropCourse(st,crs_1)} \ || \ \mathtt{enrolInCourse(st,crs_2)} \end{array}$$

and the change axioms could be stated as[12]

$$\begin{array}{l} \forall\alpha\,\forall \mathtt{st,crs}\ .\ \mathtt{(st,crs)} \in \mathtt{enrolledIn'} - \mathtt{enrolledIn} \wedge \mathtt{Occur}(\alpha) \Rightarrow \\ \qquad \alpha = \mathtt{enrolInCourse(st,crs)} \vee \\ \qquad \exists\,\mathtt{crs_1}\ .\ \alpha = \mathtt{switchCourse(st,crs_1,\ crs)} \end{array}$$

$$\begin{array}{l} \forall\alpha\,\forall \mathtt{st,crs}\ .\ \mathtt{(st,crs)} \in \mathtt{enrolledIn} - \mathtt{enrolledIn'} \wedge \mathtt{Occur}(\alpha) \Rightarrow \\ \qquad \alpha = \mathtt{dropCourse(st,crs)} \vee \\ \qquad \exists\,\mathtt{crs_2}\ .\ \alpha = \mathtt{switchCourse(st,crs,\ crs_2)} \end{array}$$

$$\begin{array}{l} \forall\alpha\forall \mathtt{crs}\ .\ \mathtt{size(crs)} \neq \mathtt{size'(crs)} \wedge \mathtt{Occur}(\alpha) \quad \Rightarrow \\ \qquad \exists \mathtt{st}.\ \alpha = \mathtt{enrolInCourse(st,crs)} \vee \\ \qquad \exists \mathtt{st}.\ \alpha = \mathtt{dropCourse(st,crs)} \vee \\ \qquad \exists \mathtt{st,crs_3}\ .(\ \alpha = \mathtt{switchCourse(st,crs,crs_3)} \vee \\ \qquad\qquad \alpha = \mathtt{switchCourse(st,crs_3,crs)}) \end{array}$$

---

[12] The exact syntax of change axioms for various specification languages should be the subject of further study.

Note that only the disjunctive clauses involving `switchCourse` are contributed by the introduction of the `switchCourse` procedure (four clauses in all).

For the object-oriented specification of Section 2.4, we explicitly name the function and put the "privileged" argument self back into the function call in order to simplify the presentation. In the case of courses with co-requisites, the specifications of actions for drop would be

```
class COURSE
  drop(self: COURSE, st: STUDENT)
     PRE:   HasEnrolled(self,st)
     POST: ¬HasEnrolled'(self,st)
```

```
class COURSE-WITH-COREQ  subclass of  COURSE
  drop(self: COURSE-WITH-COREQS, st:STUDENT)
     POST: ∀x . Coreq(self,x) ⇒ ¬HasEnrolled(x,st)
```

and change axioms for `HasEnrolled` would state

```
∀α∀st, crs
    ¬HasEnrolled(crs,st) ∧ HasEnrolled'(crs,st) ∧ Occur(α) ⇒
       α = COURSE::add(crs,st) ...

∀α∀st
    HasEnrolled(crs,st) ∧ ¬HasEnrolled'(crs,st) ∧ Occur(α) ⇒
          (α = COURSE::drop(crs,st) ∧ (crs ∉ COURSE-WITH-COREQ)∧...) ∨
          (α = COURSE-WITH-COREQ::drop(crs,st) ∧ HasEnrolled(crs,st))
```

In the case of employees moving, the specifications would be

```
EMPLOYEE.
move(self: EMPLOYEE, new:ADDRESS)
    MOD:  livesAt
    POST: livesAt'(self)=new
```

```
MARRIED_EMPLOYEE.
move(self: MARRIED_EMPLOYEE, new:ADDRESS)
    POST:  livesAt'(spouse(self))=new
```

and the change axioms for `livesAt` would state

```
∀α∀x . livesAt(x) ≠ livesAt'(x) ∧ Occur(α) ⇒
        ∃n . α = EMPLOYEE::move(x, n)
      ∨
       ∃n . α = MARRIED_EMPLOYEE::move(x, n)
      ∨
       ∃n,y . α = MARRIED_EMPLOYEE::move(y, n) ∧ x=spouse(y)
```

Finally, for the case of honors students, the specifications would look like

```
class Course
 findHonors(self:COURSE)
    PRE:  domain grade = enrolment
    POST: { s | grade(s)(self) > 85} ⊆ honors'(self)
```

```
class SENIOR-COURSE  subclass of COURSE
```

findHonors(self:SENIOR-COURSE) POST: { s │ grade(s)(self) > 80} ⊆ honors'(self)

so the conjunction of the post conditions for `SENIOR-COURSE::`*`findHonors`* would yield `{ s │ grade(s)(self) > 80}` $\subseteq$ `honors'(self).` The initial value of `honors` would be the empty set, and the effect axiom for `honors` could read

$$
\begin{aligned}
\forall\alpha\,\forall c\forall s.\ & s \in \texttt{honors'(c)-honors(c)} \ \wedge\ \texttt{Occur}(\alpha) \quad \Rightarrow \\
& \alpha = \texttt{COURSE::findHonors(c)} \ \wedge\ (\texttt{grades(s)(c) > 85}) \\
& \qquad\qquad \wedge\ \textbf{domain}\ \texttt{grade = enrolment} \\
\vee\ & \\
& \alpha = \texttt{SENIOR-COURSE::findHonors(c)} \ \wedge\ (\texttt{grades(s)(c) > 80}) \\
& \qquad\qquad \wedge\ \textbf{domain}\ \texttt{grade = enrolment}
\end{aligned}
$$

3.3 A Syntactic Technique for Generating Explanation Axioms (...Sometimes)

The astute reader may have noticed that we followed a relatively mechanistic process in building the change axioms. For the predicate `EnrolledIn`, for instance, we looked at the postcondition of each procedure p, such as *`enrolInCourse`* and *`dropCourse`*, and if it asserted `EnrolledIn'(`$\underline{w}$`)` (respectively, $\neg$`EnrolledIn'(`$\underline{w}$`)`) for some argument $\underline{w}$, then we added the clause

$(\alpha = \texttt{p}(\underline{w}))$

as an additional disjunct to the positive (respectively, negative) change axiom. For conditional postconditions with cases of the form:

$\gamma(\underline{x}) \Rightarrow \texttt{EnrolledIn'}(\underline{w})$

where γ only includes unprimed predicates and functions, we added a disjunctive clause of the form

$\exists\underline{x}(\alpha = \texttt{p}(\underline{x}) \wedge \gamma(\underline{x}) \wedge \underline{y} = \underline{w})$

where $\underline{w}$ consists of constants and variables from $\underline{x}$, to the positive change axiom for `Q`, whose antecedent is $\neg\texttt{Q}(\underline{y}) \wedge \texttt{Q'}(\underline{y}) \wedge \texttt{Occur}(\alpha)$. This process can be easily translated into an algorithm which generates change axioms. In the spirit of our earlier arguments concerning the need for flexibility, we would however only advocate the use of such an algorithm as a tool that would *assist* the specifier in adding more quickly the change axioms.

3.4 Semantic Aspects of the Approach

A problem that can arise with any purely syntactic approach, such as the one suggested in the previous section, is that it is often affected by the particular syntax chosen to express postconditions and is not necessarily self-consistent. Logicians rely on model theory, and proofs of soundness and completeness to avoid such pitfalls for their syntactic manipulations. As it happens, it is possible to provide a model-theoretic foundation for frame axioms, as explored in the literature on non-monotonic reasoning.[13]

Recall that the problem with the "loose" reading of a post-condition such as `EnrolledIn'(st,crs)` is that it does not rule out any other changes. In other words, among the models of the formula, we find some in which the student's name has changed (i.e., where `name(st)` and `name'(st)` differ), as well as ones where they do not. Model-theoretically, we would therefore like to select out those models of the postcondition in which "things changed only if they had to". In other words, differences between primed

[13] This section is somewhat more technical, and assumes that the reader is familiar with the model theory of First Order Predicate Logic.

and unprimed versions of predicates are *minimized*. in the sense that there is no model of the postcondition that has fewer changes.

For this, we can turn to the technique of circumscription [McCarthy86], which has in fact been used to attack the frame problem in AI. Circumscription essentially selects only those models of a theory which minimize the extents of certain specified predicate(s). Circumscription may be illustrated by showing how it captures the notion of "negation by failure" familiar to Prolog programmers: given the clauses

```
P(1).
Q(x):- P(x).
```

negation by failure would allow one to conclude the following list of atoms

$Q(1),\ \neg Q(2),\ \neg Q(3),\ldots$

while classical logic only entails the first atom. This is achieved by making all predicates have the smallest (positive) extent necessary to satisfy the originally given facts $\{P(1),\ \forall \underline{x}.P(\underline{x}) \Rightarrow Q(\underline{x})\}$

In our case, since we have stated that we are interested in minimizing changes, we can introduce for every predicate R two new predicate symbols $\Delta^{+}R$ and $\Delta^{-}R$, with defining axiom

$$\Delta^{+}R(\underline{y}) =_{def} \neg R(\underline{y}) \wedge R'(\underline{y}) \qquad\qquad \Delta^{-}R(\underline{y}) =_{def} R(\underline{y}) \wedge \neg R'(\underline{y})$$

Technically, we are then interested in circumscribing (minimizing) the predicates in the set $\{\Delta^{+}R, \Delta^{-}R\}$, subject to the constraint that the effect axioms of procedures hold, as do the axioms defining ΔR, and the various unique name axioms. This minimization is to be done while keeping the extents of unprimed predicates fixed but varying the extents of primed ones.

In our example, the theory would therefore have the effect axioms for `enrolInCourse`, `dropCourse` etc., such as

$$\begin{aligned}&\forall x, y[\ Occur(enrolInCourse(x,y)) \Rightarrow\\&\quad (size(y) < classLimit(y) \wedge\\&\qquad\qquad\qquad \neg(\ [EnrolledIn(x,y)])) \wedge\\&\quad (size'(y) = size(y) + 1 \wedge\\&\qquad\qquad\qquad EnrolledIn'(x,y))]\end{aligned}$$

plus the unique name axioms

$$\forall \underline{x},\underline{y}[(enrol(\underline{x}) \neq drop(\underline{y})) \wedge (enrol(\underline{x})=enrol(\underline{y}) \Rightarrow \underline{x}=\underline{y}) \wedge \ldots]$$

Appropriate circumscription using the above theory would restrict us to consider only those of its models in which the predicates $\Delta^{+}EnrolledIn$, $\Delta^{-}EnrolledIn$, $\Delta^{+}size$, etc. have minimal extents (in the sense that if any tuple is removed from any of their denotations, the resulting valuation is no longer a model of the theory derived from the specification.) The invariant would then have to be proven to be maintained only in these specially selected models.

At first glance, such a model-theoretic description would appear to be of only purely theoretical interest. However, there is the possibility of finding some syntactic formulation that can be proven to be sound, and possibly even complete with respect to it. In fact, the change axioms generated from the effect axioms by the algorithm sketched in the previous section can be shown to select out exactly the models that would be chosen by the circumscription scheme outlined in this section, for a limited class of specifications. This should increase considerably our confidence in the syntactic treatment of the frame problem discussed earlier.

4. Other Approaches to the Frame Problem

4.1. Approaches in Artificial Intelligence

The frame problem in AI has been studied within a different formal framework, called the *situation calculus* [McCarthy63], which makes explicit the relationship between the predicates describing the program state before and after the procedure -- a relationship maintained purely by the priming convention in our approach. This is done by also reifying states as follows: for each function $f(\underline{x})$ in the specification, introduce a function $f(\underline{x},s)$, called a "*fluent*", where s is a term of sort state (or *situation*). Likewise, for each predicate $R(\underline{x})$ of the specification let there be a fluent $R(\underline{x},s)$. Given this formulation, it is then possible to consider the primed-predicate notation as simply an abbreviation: in specifying procedure p, every occurrence of an unprimed atom or term $f(\underline{y})$ should be read as $f(\underline{y}, s)$, while every atom or term of the form $f'(\underline{y})$ stands for $f(\underline{y}, do(p(\underline{x}), s))$, where `do` is a special function symbol intended to the denote the state obtained by executing an action -- its first argument -- in the start state -- its second argument. The full situation calculus is therefore more expressive in that it allows one to reason about sequences of actions/procedures, e.g., "Mary is enrolled in cs2, after Mary added cs2 in state s0, and then Ann dropped cs1".

$$\mathtt{EnrolledIn(mary,cs2,\ do(drop(ann,cs1),\ do(\ added(mary,cs2),\ s_0)))}$$

Since it was first pointed out by McCarthy and Hayes (1969), the frame problem has been the subject of intense investigation in AI. During the 1980's and early 1990's, nonmonotonic reasoning has been the formal approach of choice (see (Reiter87) for a review of this field). The intuition underlying this appeal to nonmonotonic logics is that the frame axioms are the product of a form of "closed world assumption" with respect to the *causal laws* (what we have been calling "effect axioms"). In other words, it is assumed that the causal laws specified by the axiomatizer are all, and only, the causal laws for the domain, and therefore, if there is no law specifying that action `p` affects a given predicate's truth value, then that predicate must be *unaffected* by `p`. The technical problem is to provide a formal account of this closed world assumption, and since circumscription [McCarthy86] is the most general formalization of closed world reasoning, the majority of approaches to the frame problem have focussed on it [Lifschitz91], [Lin91]).

Despite this emphasis on circumscription, a few attempts were made to solve the frame problem without appealing to nonmonotonic logics, notably [Haas87], [Schubert89] and [Pednault89]. In particular, Haas and Schubert first proposed the concept of an *explanation closure axiom* (or, as we have been calling them, change axioms) for specifying the frame axioms, while Pednault was the first to formulate a mechanism for automatically obtaining the frame axioms given only the causal laws for the domain of application. By combining and extending the ideas of Haas, Schubert and Pednault, [Reiter91] provided the solution to the frame problem that underlies the proposal of this paper. This proposal was recently semantically characterized [Lin94], by showing that the circumscription described in [Lin91] characterizes Reiter's solution. Accordingly, we now have both semantic and syntactic characterizations of the frame problem in the situation calculus, and a proof of their equivalence.

An additional source of difficulties for solutions to the frame problem is the presence of state constraints (what we have called "state invariants", or what are known as integrity constraints in data bases). Unlike the stance adopted in this paper (where the specifier is expected to prove that the procedure specification is sufficiently complete to re-establish any state invariants), in AI it is considered desirable to derive from causal laws and state constraints *new* causal laws, which then affect the frame axioms. This version of the frame problem, known as the *ramification problem* , is considerably more difficult and cannot yet be said to have a satisfactory solution. For an attempt to reconcile Reiter's solution to the frame problem with the presence of state constraints, see [Lin94] .

4.2. Alternate Approaches in Software Specification

Even with appropriate language enhancements, program specifiers may forget to state the ubiquitous frame axioms. For this reason, some specification languages have opted for a richer semantics which implicitly adds the "...nothing else changes" clause to every procedure specification. In other words, such languages support an interpretation of the initial specification of `enrolInCourse` that is consistent with the "tight" reading mentioned in Section 2. The advantage of such implicit frame axioms is that if we apply them to procedure specifications *after* inheritance, conjunction, etc. have been expanded, then we get the desired effect of things not changing “unless otherwise stated”.

Several systems, such as INSCAPE [Perry89], make the meta-assertion that variables are not changed unless explicitly said so, and then implement it *operationally* in the sense that the specific reasoning done by the system take this into account. In the case of INSCAPE, the system concerns itself mostly with propagating necessary or possible changes to the interfaces, and does not support constructing proofs that invariants are maintained.

The ASLAN specification language [Auernheimer86] introduces into the language special connectives **if-then-else** and **alternate,** which have a “procedural reading” of the kind familiar to programmers. The aim is to provide an explicit (declarative) and formal expression of the intuitive statement that “nothing else changes”. The aforementioned paper offers algorithms for computing such assertions in this limited case, and provides ample evidence of the difficulties one runs into because of context and subscripted variables -- difficulties related to the meaning of the procedural reading, as well as the correctness of the implementation.

To their credit, Schuman and Pitt [Schuman87] discovered the frame problem for object-oriented specifications in Z [14], and proposed a technique for defining a "completing" assertion that can be added to a postcondition in order to capture the frame axioms. Their approach is based on the observation that in a set-theoretic notation $s' = s \cup \{1\}$ can be equivalently stated as

$$(1 \in s') \wedge (s' - \{1\} = s - \{1\}),$$

so that with a postcondition of the form $(1 \in s')$, we are looking for a weaker version of the frame axiom $s' = s$. Their scheme relies on finding maximally consistent subsets of so-called "neutral" assertions. Unfortunately, this approach suffers from the problem that it is computationally intractable (in fact, not even semi-decidable), since it relies on proving consistency of subsets of formulas.

Independently, we stumbled across the problem in our own attempts to develop a specification language specially suited for information systems. Since we were aware that expressing “nothing else changes” was one of the motivating examples of the field of Non-monotonic Reasoning, our language proposal, TaxisDL [Borgida89], explicitly states the frame axioms using Default Logic --- a particular version of Nonmonotonic Logic. Unfortunately, this scheme is also not effectively computable since it also relies on the non-provability relation.

Obviously, the non-effectiveness of the above schemes severely compromises a formal specification language with respect to its capacity to support the formal treatment of specifications, since one of the prime uses of procedure specifications is to prove that they leave state invariants intact. Such proofs can then no longer be automated.

In addition to the computational problem, another major drawback with embedding the frame axioms in the specification language semantics is that in some circumstances the specifier may want the freedom *not* to make them, or to state them in some different way. [Ryan91] and [Haugelstein92] are other recent proposals which embed *some* frame axioms in the semantics of the specification language by allowing the specifier to differentiate actions or attributes for which the assumptions are to be made from those for which they are not. A different compromise between placing the full responsibility on the specifier, on one hand, and embedding the frame axioms in the semantics of the specification language, on the other, is to provide a computer tool which, once a specification is considered finished, makes a pass over it, expands abbreviations, and suggests frame axioms to be added to each specification. If the user is unhappy with the proposed axioms, she can modify them to her liking. At the very least, such a tool would be very helpful in catching omissions. It must be stressed however that such a tool ought to be viewed as a compiler -- when changes are made to the specification, the tool needs to be re-run. The work in [Penny90], and to some

[14] Regrettably, their insights seem to have been ignored by more recent papers on object-oriented specifications.

extent [Auernheimer86], exemplifies this approach, though it only deals with a restricted subset of formulas. Our proposal for syntactically generating change axioms can be viewed in the same light.

5. Concluding Remarks

We have presented evidence that in specifying procedures in the popular style of pre- and postconditions defined over program states, it is useful to have some mechanism to make blanket statements of the form "...and nothing else changes", rather than explicitly enumerating the things that do not change. This facility would seem to be essential if we are to develop a proper notion of object-oriented specifications to support programming in OOPLs such as C++ and Eiffel.

We critically reviewed some of the mechanisms in extant specification languages that attempt to address this problem. Secondly, we have applied a technique that has been recently used in AI to deal with the frame problem: the idea of axioms explaining what changes could occur to a particular component of the program state through the application of known procedures. This idea, we have argued, produces short, effective and relatively modular specifications for software, and deals effectively with a number of problems that are not properly addressed by other approaches. Moreover, our proposal has the added advantage of being phrased entirely within Predicate Logic, thus making it unnecessary to build special theorem provers to reason with specifications. We also note that the specialized context of reasoning about procedure specifications as currently performed in software engineering makes it possible to simplify considerably the machinery used in AI.

We have also argued that for a small class of specifications, it is possible to generate syntactically some explanation closure axioms that coincide with those suggested by our intuitions, and that these intuitions are supported by a model-theory in which the extents of predicates are minimized.

The following are just some of the issues that remain to be explored:

- How large is the class of specifications for which we can generate automatically reasonable change axioms starting from pre/postconditions?

- What are appropriate notations that allow these ideas to be incorporated into familiar specification languages such as Z, VDM, Larch, COLD, etc. without violating or extending too much their underlying philosophy?

- The explicit inclusion of frame axioms into the specification also causes difficulties with reasoning about concurrency in specification, as observed by [Shuman87]. Our approach can form a starting point for exploring new approaches, by allowing the `Occur` predicate to hold at more than one argument/action at a time.

- In some circles, it has been suggested that in order to support maximal reuse, there should be no limitation on the kinds of changes that one can make to a procedure's specification when specializing/refining it to a subclass. The wider use of `Occur` predicate in post-conditions may allow for the explicit cancellation of inheritance.

Acknowledgment: We are grateful to Eric Hehner for his comments and much appreciated moral support, to Klaus Dieter Schewe and Don Cohen for clarifications concerning certain formal specification systems, and to the reviewers.

References

[Alencar94]
A. J. Alencar and J. A. Goguen, "Specification in OOZE with Examples", in [Lano94], pp. 158-183.

[Auernheimer86]
A. Auernheimer and R. Kemmerer, “Procedural and Nonprocedural Semantics of the ASLAN Formal Specification Language”, *Ptoc. 19th Hawaii Int. Conf. on System Sciences (HICS),* 1986, pp.784--794.

[Borgida82]
A. Borgida, "On the definition of specialization hierarchies for procedures”, *Proc. IJCAI’81,* Vancouver, BC, August 1981, pp.254--256.

[Borgida84]
A. Borgida J. Mylopoulos, and H. K. T. Wong, "Generalization/Specialization as a Basis for Software Specification", in *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases and Programming Languages*, M. Brodie, J. Mylopoulos and J. Schmidt (eds.), Springer Verlag, 1984, 87-114.

[Borgida90]
A. Borgida, J. Mylopoulos, J. Schmidt and I. Wetzel, "Support for Data-Intensive Applications: Conceptual Design and Software Development", in *Database Programming Languages*, R.Hull, R. Morrison, D.Stemple (eds.), Morgan Kaufmann Publishers, San Mateo CA., 1990.

[Cohen86]
B. Cohen, W.T.Harwood, and M.I.Jackson, *The Specification of Complex Systems,* Addisson Wesley, 1986

[Duke90]
Duke, D. and R. Duke, "Towards a Semantics for Object Z", in *VDM and Z --- Formal Methods in Software Development*, Lecture Notes in Computer Science, Vol. 428, 244-261, Springer Verlag, Berlin, 1990.

[Guttag85]
J. Guttag, J.J. Horning, and J.W. Wing, "Larch in Five Easy Pieces", TR 5, DEC Systems Research Center, 1985.

[Haas87]
Haas, A. R., "The Case for Domain Specific Frame Axioms", in Brown, F. M. (ed.) , *The Frame Problem in Artificial Intelligence. Proceedings of the 1987 Workshop*, 343-348, Morgan Kaufmann Publishers, Inc., 1987.

[Hall90]
A. Hall, "Using Z as a Specification Calculus for Object-Oriented Systems", *VDM and Z --- Formal Methods in Software Development*, Lecture Notes in Computer Science, Vol. 428, 290-317, Springer Verlag, Berlin, 1990.

[Haugelstein92]
J. Haugelstein and D. Roclants, "Reconciling Operational and Declarative Specifications", *Proceedings International Conference on Advanced Information Systems Engineering (CAiSE)*, Manchester, May 12-15, 1992.

[Jones86]
C.B. Jones, *Systematic Software Development Using VDM*, Prentice Hall, London, 1986.

[Jonkers91]
H.B.M.Jonkers, "Updating the pre- and postcondition technique", in *VDM'91 Formal Software Development Methods,* Lecture Notes in COmputer Science, Springer Verlag, 1991, pp.428-456.

[Lanno94]
K. Lanno and H. Haughton, *Object-oriented Specification Case Studies*, Prentice Hall, 1994.

[Lifschitz91]

Lifschitz, V., "Toward a Metatheory of Action", Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR'91), 376--386, Morgan Kaufmann Publishers, Inc., 1991.

[Lin91]
Lin, F., and Shoham, Y., "Provably Correct Theories of Action", Proceedings of the National Conference on Artificial Intelligence, 1991.

[Lin94]
Lin, F. and Reiter, R., "State Constraints Revisited*", Journal of Logic and Computation 4*, 655--678, 1994.

[Marshall91]
L. Marshall and L. Simon, "Using VDM within an Object-Oriented Framework", *VDM and Z --- Formal Methods in Software Development*, Lecture Notes in Computer Science, Vol. 428, Springer Verlag, Berlin, 1991

[McCarthy63]
J. McCarthy, "Situations, Actions, and Causal Laws", technical report, Stanford University, 1963; also reprinted in *Semantic Information Processing*, Minsky, M., (ed.), MIT Press, 1968.

[McCarthy69]
J. McCarthy and P. Hayes, "Some Philosophical Problems from the Standpoint of Aritificial Intelligence", *Machine Intelligence 4*, Melzter, B. and Michie, D.(eds.), 463-502, Edinburgh University Press, 1969.

[McCarthy86] McCarthy, J., "Applications of Circumscription to Formalizing Commonsense Knowledge", *Artificial Intelligence 28*, 89-116, 1986.

[Meyer88]
B. Meyer, *Object Oriented Software Construction*, Prentice Hall, 1988.

[Mylopoulos80]
J. Mylopoulos, P. Bernstein, and H. Wong, "A Language Facility for Designing Data-Intensive Applications", *ACM TODS 5(2),* June 1980.

[Pednault89]
Pednault, E.P.D., "ADL: Exploring the Middle Ground Between STRIPS} and the Situation Calculus", Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89), 324--332, Morgan Kaufmann Publishers Inc., 1989.

[Penny91]
D. Penny, R. C. Holt, M. Godfrey, "Formal Specifications in Metamorphic Programming", *VDM'91 -- Formal Software Development Methods,* Lecture Notes in Computer Science, Springer Verlag, 1991, pp.11-30.

[Perry89]
D. Perry, "The Inscape Environment", *Proc. 11th Int. Conf. on Software Engineering*, Pittsburgh, May 1989.

[Reiter87]
R. Reiter, "Nonmonotonic Reasoning", *Annual Review of Computer Science 2*, 147-186, Annual Reviews Inc., 1987.

[Reiter91]
R. Reiter, "The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression", in V. Lifschitz, (ed.), *Artificial Intelligence and the Mathematical Theory of Computation: Papers in Honor of John McCarthy*, 359-380, Academic Press, San Diego, CA, 1991.

[Reiter95]
R. Reiter, "On Specifying Database Updates", *Journal of Logic Programming* (to appear).

[Ryan91]
M. Ryan, J. Fiadeiro, and T. Maibaum, "Sharing actions and attributes in modal action logic", in *Theoretical Aspects of Computer Software*, 569-593, Springer Verlag lecture Notes in Computer Science, no. 526, 1991.

[Schewe91]
K-D. Schewe, J. W. Schmidt, and I. Wetzel, "Specification and Refinement in an Integrated Database Application Environment", *VDM'91 -- Formal Software Development Methods,* Lecture Notes in Computer Science, Springer Verlag, 1991, pp.496-510.

[Schubert89]
Schubert, L. K., "Monotonic Solution of the Frame Problem in the Situation Calculus: An Efficient Method for Worlds with Fully Specified Actions", in Kyberg, H. E., Loui, R. P. and G.N. Carlson (eds.), *Knowledge Representation and Defeasible Reasoning*, 23-67, Kluwer Academic Press, 1990.

[Schuman87]
S.A. Schuman, and D.H.Pitt, "Object-Oriented Subsystem Specification", in *Program Specification and Transformation*, L.G.L.T.Meertens (ed), North Holland, 1987, 313-341.

[Spivey89]
J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall, 1989.

[Stepney92]
S. Stepney, R. Barden and D. Cooper (eds.) , *Object Orientation in Z,* Workshops In Computing Series, Springer Verlag, 1992.