The frame problem in object-oriented specifications: an exhibition of problems and approaches *

Alex Borgida
borgida@cs.rutgers.edu

Technical Report LCSR-TR-209

Dept. of Computer Science
Rutgers University
New Brunswick, NJ 08903

October 2, 1992

Abstract

We present first a series of examples involving the development of information systems, which suggest a number of desirable features for object-oriented specification techniques, especialy those supporting inheritance. Most of these features have difficulties with the so-called *frame axioms* — assertions which state what values have been left unchanged by some procedure.

We then examine the benefits and disadvantages of a variety of proposals for dealing with the frame problem, some of which are based on ideas presented earlier in the literature, while others are novel. The approaches are grouped into two families: one which introduces notational conventions/abbreviation for stating frame axioms, and one which embeds them into the language semantics.

Of particular interest may be the introduction of a model-theoretic version of the assumption that things don't change unless they have to, and the possibility of relating this to syntactic techniques for stating such frame axioms in standard logic.

*This research was supported in part by grants from the National Science and Engineering Research Council of Canada, the Institute for Robotic and Intelligent Systems (IRIS) (funded by the Government of Canada through the Networks of Excellence programme), and the NSF of USA under grant IRI91-19310.

1 Introduction

There is considerable interest in object-oriented approaches both in the general programming community, and in the area of formal program specification, as evidenced by the profusion of papers on this topic at recent conferences, including the following papers in the Z paradigm [26, 12, 15, 2, 16].

In this paper, we shall concentrate on the specification of a particular family of software systems, namely Information Systems.[1] Information systems maintain models of some aspect of the "real world" by storing information about individuals, their inter-relationships, and the activities that change these relationships. Such models are then "queried" through other operations in order to obtain information necessary to make decisions, provide statistics, etc. It is important that such models be as "natural" as possible to end-users, in the sense that they should correspond to their conceptualization of the application domain.

Information systems are usually implemented using some sort of database management system, and have two properties of relevance to our discussion: by some measures they form the single most significant class of practical computer applications; and they are frequently quite large — some CAD systems, for example, maintain information about tens of thousands of different *kinds(classes)* of parts, as well as individual parts numbering in the hundreds of thousands or millions.

The history of object orientation in Information System development is quite long, beginning with the so-called conceptual/semantic modeling languages such as Taxis [21], which were often based on knowledge representation ideas. (See [5] for a review and comparison of such languages.) More recently, several object-oriented databases, supporting persistence for object-oriented languages, have been introduced, O_2 [22] and Gemstone [9] being just two exemplars.

However, there has been relatively little work in Information Systems on specification languages for describing *what* is to be implemented, rather than programming languages describing how things are to be implemented. This paper is particularly concerned with specification languages that describe procedures in the familiar pre- and post-condition notation, relating the values of "variables" in the starting and terminating states of the procedure. In trying to define one such language, TDL [7], we encountered serious difficulties because the obvious way of describing "more specialized procedures" — a key notion of object-oriented approaches with inheritance — lead to inconsistencies. We identified the need to state explicitly that variables do not change — something which is not done in programming languages — as the source of our difficulties, and then discovered that these problems in fact arise in some form or another in all kinds of specifications, as explained in [8].

Surprisingly, with rare exceptions (e.g. [26, 23]), the work in the Formal Methods community has not addressed directly these issues. This paper is therefore intended to first of all present some examples of situations where the need to express explicitly the so-called "frame axioms" leads to difficulties. Afterwards we propose to briefly look at a whole series of possible approaches to the problem: one group is based on introducing special notation for asserting frame axioms; another, builds in to the semantics of the specification language a kind of inertia, in the sense that variables have the same value in the starting and final

[1] We shall however point out that our remarks apply to object-oriented specifications in general.

states, unless this leads to contradiction.

2 Motivating Examples

As usual, we view the specification to be a description of the desired software system that is more clear and understandable than the eventual implementation. Its role is two-fold: to act as a communication medium between programmers and user requirements; and to be reasoned with formally, in order to prove such properties as (internal) consistency and non-violation of invariants, or to answer "what-if" questions (including rapid-prototyping simulation). In addition, we must keep in mind that specifications must also facilitate reuse and modification, since maintenance is the most expensive part of software development.

2.1 Plain object-oriented specifications

Based on past research, the preferred technique for specifying the information system is by building a conceptual model of the domain consisting of the *entities* involved, and their *relationships*. This leads to an object-oriented specification, where we describe various classes of objects (e.g., `PERSON`) in terms of the attributes (binary relations) relating their instances (e.g., `name, parents, children, address`)[2]. One then describes operations (transactions) which model activities in the real world and update the state of the model (e.g., `Hire, ChildBorn, Move`).

For example, the following might be part of the specification for the class `PERSON`

```
CLASS PERSON WITH
ATTRIBUTES
  name : STRING
  age : INTEGER
  children : SET OF PERSON
  homeAddress : ADDRESS
  ...
INVARIANTS
  : (name IS NOT NULL)  ;; name is a total function over Persons
  : (age < 120)
  ...
METHODS
  Move(newAddress? : ADDRESS)
    [pre  : true    ;;precondition
     post : homeAddress' = newAddress? ]
  Hire(maritalStatus? : {1,2,3,4,5},...)
    [pre  : ...
     post : ... ∧ (maritalStatus?=1   ⇒    process single person)
                ∧ (maritalStatus?=2   ⇒    process married with no children)
```

[2] We will use examples from a hypothetical Information System dealing with a clinic which treats patients and their families.

$\wedge$ ...]

Note that procedure specifications may involve conditioned actions, as in `Hire`, and may be nondeterministic, as in the case of query actions that return random samples of data.

Note also that we might want to have invariants (called integrity constraints in databases) that involve the properties of several objects, e.g., to specify that a person and their spouse cannot have the same gender.

The frame problem rears its head:
The state of the database is recorded in this case by the attribute values for the objects that have been created, and by the "extents" associated with every class, which keep track of the current set of objects asserted to be in that class (e.g., class `PERSON` has an associated set `Persons`).

Unfortunately, according to normal conventions the above specification of `Move` permits any additional changes to other state variables, for example changing the name or the age of the person. To see this, note that we cannot use the post-condition of `Move`, POST_Move, to prove the conjecture that the invariant about ages is maintained:

$$(age < 120) \wedge \text{POST\_}Move \;\; \Rightarrow \;\; (age' < 120)$$

The above assertion does however become easily provable in standard first order logic once we add to POST_Move the predication $(age' = age)$. Of course, we need to add similar **frame axioms** for the other variables of the class, such as `name`.

Without any further help, the result is a much more complex specification for `Move`. Moreover, changing the class definition by adding another attribute, `salary` say, requires one to revise the specification of `Move`, and all other methods, by adding the conjunct $\wedge\, (salary' = salary)$ to their post-condition. Thus changes are not localized to methods directly affected.

Finally, conditional specifications, such as the one for `Hire`, need to contain explicit assertions for each sub-case, stating that variables modified in other branches of the conditional are not changed in this one (e.g., $((children' = children)$ in case `maritalStatus` is 1).

2.2 The global state

Some object-oriented approaches emphasize the encapsulation of implementation decisions and therefore hold that the attributes introduced in a class should only be visible to its own methods. We find this argument less convincing in the case of specifications, where one is not supposed to talk about implementations, and hence there ought to be no decisions to hide. Moreover, there are good reasons why in information systems the relationships between individual objects (expressed through the attributes) should be globally visible and modifiable. For example, if we wanted to define a method `ParentsOld?`, which checks if the parents of the person are old, it would have to look at the values of the `birthdate` attribute of the parents, and also verify that they are still alive. Or if we wanted to distinguish moving out of a house (as when parents separate) from moving with the family, then we may want

to have method `Move` change the address of not only the person, but also his/her children, as indicated by the following post-condition

$$homeAddress' = newAddress? \wedge$$
$$(\forall x)x \in children \;\Rightarrow\; x.address' = newAddress?$$

In order to access the `address` field of children we use the "dot" notation, which can be explained by viewing all attributes as binary relations/predicates[3] (those which have type `SET OF` are arbitrary relations, while the others are one-to-one) and then interpreting $x.address' = newAddress?$ as the formula $(\forall y)address'(x,y) \;\Rightarrow\; (y = newAddress?)$.

To make the notation fully uniform, it is then more proper to introduce a variable $self$, ranging over the individuals in the class (and an implicit parameter of all procedures), writing the above post-condition as:

$$self.homeAddress' = newAddress? \wedge$$
$$(\forall x)x \in self.children \;\Rightarrow\; x.address' = newAddress?$$

Finally, it has been found very convenient in the database area to extend this notation to allow "nested dot expressions" so that (`self.age` < `self.mother.age`) is an abbreviation for

$$((\forall m,a,b)\texttt{age(self,a)} \wedge \texttt{mother(self,m)} \wedge \texttt{age(m,b)} \;\Rightarrow\; (a < b))$$

Using this abbreviation, the change of address becomes simply

$$self.homeAddress' = newAddress \wedge self.children.address' = newAddress$$

The frame problem gets worse:
In this case, the frame problem is made worse by the fact that we need to also assert that the `dependent`'s name, etc. did not change, though their `addresses` did. For this we need to add assertions such as

$$\wedge\ self.children' = self.children$$
$$\wedge\ self.children'.name' = self.children.name$$
$$\wedge\ self.children'.age' = self.children.age$$

Since procedures can now affect the values of all state variables, we must repeat the above for every possible individual PERSON, and in addition assert that addresses are changed only for the children of the person:

$$(\forall y)(y \in Persons \;\wedge\; y \notin self.children) \;\Rightarrow\; y.address' = y.address$$

In fact, we now need to explicitly assert that no class extents and no other attributes of other objects have changed, which is an enormous task for a large database system.

We remark that this problem arises for all ordinary procedural object-oriented languages such as C++, Smalltalk, or Eiffel, where in executing a method `p` for an object `b`, one can "send a message" `q` to some object `c` stored in a local variable of `b`, and this message may change the state of `c`. Therefore the execution of a method on some object may, through side effects, change the state of arbitrary other objects. Note also that this is

[3] A similar interpretation of Z-like schemas into first order logic is suggested in [29].

not just some example of "bad style" but the essence of the paradigm of "message passing between independent agents", which is touted as one of the important advantages of object orientation.

2.3 Subclasses and inheritance

Another property of object-oriented specifications is that they support factoring out commonalities between many classes into a superclass, whose description is then "inherited" so that it need not be repeated.

For example, if PATIENT is defined as follows:

```
CLASS PATIENT IS A PERSON, . . .
   ATTRIBUTES
      disallowedFoods : SET OF FOOD
      . . .
     METHODS
      TakeHistory
      . . .
```

then instances of PATIENT have all the attributes and methods of PERSON, plus the additional ones mentioned in its definition. Furthermore, the invariant stating that

$$(Patients \subseteq Persons)$$

is automatically added to the specification.

The subclass hierarchy and inheritance are useful for a number of reasons (see for example [4, 6]), including:

- Abbreviation: the specification of the entire system is made shorter (hence more easily readable) by factoring out commonalities.
- Propagation of changes: when a super class is changed (e.g., by the addition of a new attribute or by changing the post-condition of a method), all its subclasses inherit the change.
- Reuse and extensibility: the ability to add new subclasses facilitates software maintenance by encouraging the writing of variants rather than complete redevelopment.
- Polymorphism: procedures developed for objects of one type, can be applied to all objects whose type is a subtype of the original one.
- Verification: in some situations, it is possible to re-use proofs of correctness so that they are inherited to subclases (see [3]).

It is widely agreed that during specialization new attributes may be added to a class, as well as new methods modifying them. There are however a number of extensions of this idea which are potentially very valuable:

1. In many object-oriented languages, such as Eiffel and Taxis, it is possible to create objects which are instances of classes that may themselves have subclasses. Thus there may be instances of `PERSON` that are not in any subclass of `PERSON` (yet)[4].

2. The methods of a subclass may need to change the attributes introduced in superclasses. For example, if `HEART-PATIENT` is a subclass of `PATIENT`, then the method `HEART-PATIENT$restrictDiet` needs to add high-cholesterol foods to the set of `disallowedFoods`, which was introduced in the superclass `PATIENT`.

3. The subclass may not only add new attributes but also specialize the constraints on inherited attributes (e.g., ages of `PERSON`s are less than 120, but ages of `EMPLOYEE`s are less than 75).

4. The subclass may also specialize previously introduced methods. One of the two distinguishing features of object-oriented programming languages (as noted in [27]) is in fact the ability to dynamically select the appropriate version of the procedure to be executed, depending on the class membership of the individual object: `p.restrictDiet` will execute different procedures depending on the class to which `p` belongs.

5. In languages such as Eiffel [20] and Taxis [21] specialization of a method is restricted to strengthening the post-condition[5]. For example, `HEART-PATIENT$admit` does additional things to those performed by `PATIENT$admit`, such as notifying the cardiology resident on call. The reason for requiring the strengthening is in order to be able to view the execution of `HEART-PATIENT$admit` as also an execution of `PATIENT$admit`. This allows us to state the general rule that if some individual `b` is in class `C`, with method `p`, then the execution of `b.p` satisfies the post-condition of `C$p`, whether or not `b` is also an instance of some subclass of `C`. (See also item 1 in this list.)

6. The strengthened post-condition may need to effect the same attribute modified by the more general method. For example, `restrictDiet` for `HIGH-BLOOD-PRESSURE-PATIENT`, which is a subclass of `HEART-PATIENT`, also needs to add salted foods to the set of `disallowedFoods`, which already had added to it high-cholesterol foods by `HEART-PATIENT$restrictDiet`.

We summarize and schematize the above points in Figure 1, which shows a class `B` with attributes/local variables `p` and `q`, and a method `m`, together with several subclasses, each of which specializes `m` in different, but problematic ways. In each case, we have omitted the frame axioms, whose expression will be the main concern of the remainder of the paper.

The frame problem becomes acute:

For the purposes of polymorphism, inheriting changes and software reuse, as well as many of the points above, it seems that the natural way to proceed is to define inheritance by conjoining the specification of the superclass with that of the subclass, and in the process conjoin the post-conditions of specialized methods. In this case, if we explicitly state the

[4]Just like there are rectangles that are not squares in the standard polygon hierarchy used to motivate object-orientation.

[5]There is debate whether pre-conditions should be weakened or strengthened during specialization. For simplicity, we will side-step this issue here by assuming that all preconditions are true.

```
CLASS B WITH
  ATTRIBUTES p,q : SET OF INT;
  METHODS [POST_m:  p'=p ∪ {1}]


   CLASS D IS A B WITH
     ATTRIBUTES r : SET OF INT;
     METHODS [POST_m:  q'={3}   ;;  as in point 5]

   CLASS E IS A B WITH
     ATTRIBUTES ...
     METHODS [POST_m:  p'= p ∪ {4} ;;  as in 6]

   CLASS F IS A B WITH
     ATTRIBUTES s : B
     METHODS [POST_m:  s.p' = {5} ;;  global change]

   CLASS G IS A B WITH
     ATTRIBUTES s : INT, w : INT
     METHODS [POST_m: (w < 2  ⇒  q' = {6}) ∨ (w > 5  ⇒  s' = {6})]
                    ;;  disjunctive and conditioned goal
```

Figure 1: Schematic subclasses illustrating various kinds of method specialization

frame axioms in each post-condition then all specializations of a method become inconsistent: the specification of `PATIENT$admit` says that nothing else changes than what has just been said (including the cardiology resident's status), but `HEART-PATIENT$admit` goes and modifies just this status.

Note that the "trick" of not stating frame assertions on `PATIENT$admit` — leaving them to be stated only on the leafs of the subclass hierarchy does not work when, as claimed above, we want to have objects in `PATIENT` which are not in any subclasses. And defining two classes, `PATIENT*` — which is a dummy class, used only for inheritance purposes — and a new subclass, the 'real' `PATIENT`, is not acceptable since it destroys the "naturalness" of the domain model (which we claimed in the introduction, was an important consideration in building Information Systems).

3 Notational solutions to the frame problem

We examine a number of techniques for dealing with the above problems, some of which are based on ideas presented elsewhere. Since there appears to be no "best" general solution, in each case, our goal is to present the problems which can and cannot be resolved using the technique in question.

Most of the approaches in this section do *not* attempt to deal with problem 6, illustrated in the definition of class E. Also, we point out that all solutions below treat attributes as single, local variables. These approaches can be modified to deal with the problem of global variables, illustrated in Section 2.2, by viewing attributes as global binary relations, so that a simple specification of the form [POST_m: $p' = p \cup \{1\}$], becomes [POST_m(self): $p' = p \cup \{(self, 1)\}$] or [POST_m(self): $p'(\vec{y}) \iff (p(\vec{y}) \vee \vec{y} = (self, 1)]$.

3.1 Separating frame and effect axioms

This relatively straight-forward approach suggests first separating the post-condition into two parts: the *intended effects* of the action, and the *frame assertions* needed to keep out unwanted changes. For example, procedure `B$m` would be specified by the pair of assertions $\langle (p' = p \cup \{1\}), (q' = q \wedge r' = r \wedge ...)\rangle$, while `D$m` has the expression $\langle (q' = \{3\}), (r' = r \wedge ...)\rangle$.

The rule of inheritance would then be expressed as follows: *effects are inherited and conjoined, but frame assertions are not inherited; they need to be written anew in each case.*

The final goal of a procedure is obtained by conjoining the effect and frame axioms. For example, after inheritance the post-condition of `D$m` would be $(p' = p \cup \{1\} \wedge q' = \{3\} \wedge r' = r \wedge ...)$.

This (straw-man) approach resolves standard inconsistencies, and could be made to deal even with disjunctive specifications, by using conditions in the frame axioms: e.g., for `G$m`,

$$\begin{array}{l}\langle (w < 2 \;\Rightarrow\; q' = \{6\}) \vee (w > 5 \;\Rightarrow\; s' = \{6\}),\\ \;(\neg(w < 2) \;\Rightarrow\; q' = q) \wedge (\neg(w > 5) \;\Rightarrow\; s' = s)\rangle\end{array}$$

The main disadvantages of this approach are that it does not deal with the problem of verbosity, and makes inheritance less effective since frame axioms need to be rewritten every

time.

3.2 Context-independent abbreviation

The standard approach in languages such as Z or VDM is to introduce some abreviatory schema for expressing the frame axioms. Following ObjectZ [12] (with its $\Delta(p)$ notation indicating that everything but p remains unchanged), and Larch [14] (with its MODIFIES AT MOST assertion), we use the notation Δ(x,y) to indicate that "everything but maybe x and y remain unchanged".

Continuing with the previous approach — where we separate effect and frame assertions — the resulting specification for `B$m` would be

$$\langle (p' = p \cup \{1\}), \Delta(p) \rangle$$

while for `D$m` we would write

$$\langle (q' = 3), \Delta(p, q) \rangle$$

Note that in order to deal with global updates, the expansion of the notation Δ(p) would have to mention every variable in the global state of the program. One way to define Δ(p) in a notation like Z would be to make an explicit assertion that *everything* remains unchanged and then use the hiding operator to hide the names of p and p'.[6]

In order to deal with disjunctive effects, we need to allow frame assertions which use $\Delta()$ inside complex formulas, so that we could write the following specification for `G$m`

$$\begin{array}{l} \langle (w < 2 \;\; \Rightarrow \;\; q' = \{6\}) \vee (w > 5 \;\; \Rightarrow \;\; s' = \{6\}) \;\; , \\ \quad (w < 2 \;\; \Rightarrow \;\; \Delta(q)) \wedge (w > 5 \;\; \Rightarrow \;\; \Delta(p)) \wedge (\neg (w < 2 \vee w > 5) \;\; \Rightarrow \;\; \Delta())) \rangle \end{array}$$

Note that languages such as Larch and ObjectZ currently do not appear to allow such nested $\Delta()$-like schema occurrences — $\Delta()$ shows up only at the beginning of the procedure specification.

3.3 Abbreviation with inheritance

One drawback of the preceding approach is that since frame assertions are not inherited, we need to mention among the arguments of Δ all the variables modified by the same procedure in various *super*classes, which leads to non-locality and non-inheritance of modifications. We could try to deal with this by introducing special inheritance rules for frame axioms, so that if the specification of `B$m` was $\langle (p' = p \cup \{1\}), \Delta(p) \rangle$, while that for `D$m` was $\langle (q' = 3), \Delta(q) \rangle$, then after expanding inheritance, the full specification for `D$m` becomes

$$\langle (p' = p \cup \{1\} \wedge q' = 3) \;\; , \;\; \Delta(p) \wedge \Delta(q) \rangle$$

and the general rule is that $\Delta(p) \wedge \Delta(q) \equiv \Delta(p, q)$. Note however that in the case of disjunctive specifications illustrated in the previous section, this technique no longer works.

[6] This approach was suggested to me in personal communication by James Power, Dublin City University.

3.4 Context-dependent abbreviation

If we take the attitude that a procedure can only see the locally declared attributes and those of its superclasses — as in ObjectZ [12, 10] and Z++ [16], then Δ(p) can be expanded context-dependently, depending on the visibility of the variables. For example, in class `B`, Δ(p) asserts $(q' = q)$, while in methods of class `D` it says $(q' = q \wedge r' = r)$.

Unfortunately, this approach still does not deal with examples such as those of classes `D` and `E` in Figure 1.

3.5 Predicate transformers

We note here that specification techniques based on predicate transformers (e.g., [11, 1]) appear to sidestep the frame problem by bottoming out in some kind of assignment/substitution statement of the form `[ v := E]`, where E can involve non-deterministic choice. Since this predicate transformer leaves unchanged every part of an assertion other than the variable `v`, it seems to solve the frame problem. (Interestingly, the original paper on the frame problem by McCarty and Hayes [18] listed the assignment statement as one possible avenue to be explored.)

One approach to solving the frame problem is then to try to convert the specification of the goal (without frame assertions) into a predicate transformer/generalized substitution. This can be done, for example, using Abrial's `@x.P` construct, which is to be read as *"choose random x satisfying predication P"*. For example, a post-condition such as $x' = 1 \vee z' \neq 3$ would lead to the predicate transformer

$$[@w_1.@w_2.(w_1 = 1 \vee w_2 \neq 3) \quad \Rightarrow \quad x := w_1 \ [] \ z := w_2]$$

which is to be read as *"choose random* w_1, w_2*, satisfying* $(w_1 = 1 \vee w_2 \neq 3)$*, and substitute these for* `x,z`*"*.

This approach was taken in the FIDE project [25], where specifications in TDL were replaced by an "equivalent" abstract machine specification [1]. One major hurdle faced by this technique is deciding over what set of variable names should the substitution take place. For example, if we had another variable `p` that could vary at will, we would write the predicate transformer as

$$[@w_1.@w_2.@w_3.(w_1 = 1 \vee w_2 \neq 3) \quad \Rightarrow \quad \texttt{x} := w_1 \ [] \ \texttt{z} := w_2 \ [] \ \texttt{p} := w_3]$$

A natural convention to assume is that we use only the set of variable names appearing in the goal assertion, so that other things remain unaffected. This has the possibly unsettling feature that adding a tautology to the postcondition affects the procedure specification: if instead of $v' = 1$ we say $v' = 1 \wedge (z' = 2 \vee \neg(z' = 2))$, then the second post-condition is interpreted to mean that z may be changed in an arbitrary way, while the first one does not permit an implementation that modifies z.

4 Frame axioms through language semantics

None of the above techniques, except for the one introduced in Section 3.2, appears to be able to deal with the problems raised by specializations like class `E` in Figure 1, which require *additional* modifications to composite (set-valued) variables introduced in superclasses. As

argued in [26], the problem can be attributed to an over-specification: all we intend to say in `B$m` is that 1 is added to the set in attribute `p`, and then `E` asserts that 4 is also added. The key idea introduced in [26] is that goals such as $(p' = p \cup \{1\})$ and $(p' = p \cup \{4\})$ in fact act in part as frame axioms for set-valued variables, and should instead be rephrased as $(1 \in p')$ and $(4 \in p')$ respectively; their conjunction then becomes consistent.

This corresponds to converting the post-conditions from being descriptions of the desired final state to being descriptions of the *changes* that need to be achieved in order to reach the final state.

We must however find ways to prevent p', and the other variables, from changing in other, unwanted ways. There are several approaches to this, most of them relying on an extension of the language semantics so that the frame assertion are automatically derived from the statement of the procedure effects. The advantage of building the frame assumptions into the language semantics is that it allows all specifications to be abbreviated.

4.1 Default reasoning

The intuition behind the approaches which view the specification as talking about what has changed, is the assumption that what has not been mentioned remains unchanged. As noted in [8], the above effort to state that things have not changed unless otherwise provable, is one of the topics studied by non-monotonic logic (see, for example, the collection of papers in [13]). In the specification language TDL [7], we used default logic [24] to express that things tend to remain as they are. This is done by asserting that for every value `c` and set-variable `p`, if `c` is (is not) in `p`, and it is consistent to assume that `c` is (is not) in `p'`, then do so. Formally, this is expressed as the set of default inference rules $\frac{x \in p \,:\, x \in p'}{x \in p'}$ and $\frac{x \notin p \,:\, x \notin p'}{x \notin p'}$ — one for every attribute p.

Unfortunately, this scheme relies for reasoning on the proof theory of default logic, which is not even semi-decidable. Also, it is known that such default theories have multiple distinct extensions, so that we need to find the theorems that hold in all of them.

4.2 "Neutral" predications

The idea here is suggested by the following equivalence:

$$p' = p \cup \{1\} \quad \equiv \quad (1 \in p') \wedge (p'/\{1\} = p/\{1\})$$

This means that with post-conditions such as $(1 \in p')$ we are looking for weaker versions of the frame axioms $p' = p$; these weakenings need to be as strong as possible, yet still be consistent with the post-condition.

According to Schuman and Pitt [26], the appropriate weakenings come from the set of formulas

$$Neutral(V, POST) = \{\alpha | (\textstyle\bigwedge_{w \in V} w = w') \;\Rightarrow\; \alpha, \\ \alpha \text{ consistent with } POST\}$$

where V is the set of all variables in the program state.

For set-theoretic attributes like `p`, such weakenings are claimed to be characterized by assertions of form $(p'/w = p/w)$ for some set w.

For disjunctive specifications such as $x' = 1 \vee z' = 3$, Neutral({x,z}) contains both x'=x and z'=z but not their conjunction. Following the spirit of [26], we must therefore look at the maximal sets of neutral formulas consistent with the post-condition, and to be cautious, we should take as frame axioms only those assertions that are in the intersection of *all* of them, i.e., can be consistently believed in every possible terminating state of the procedure.

Of course, this is a non-effective specification of the frame axioms that need to be added to every post-condition expressing the effect of the procedure, and in general, no practical reasoning systems can be built to do this. In the case of restricted kinds of post-conditions, it may however be possible to generate automatically the frame axioms.

4.3 Change-explanation axioms

We can obtain a variant of the above approach as follows[7]: The axiom stating that predicate R did not change is written as $(\forall \vec{y}) R(\vec{y}) \iff R'(\vec{y})$. Since we want to consider weakenings of these formulas, let us allow guard conditions on the values $\vec{y}$ for which the above equivalence holds. In other words, consider as candidates for neutral predications those that are conjunctions of formulas of the form $(\forall \vec{y}) \Psi(\vec{y}) \Rightarrow (R(\vec{y}) \iff R'(\vec{y}))$ for some formula Ψ. For abbreviation, let us introduce the new predicate ∇R, indicating that R and R' are different, which is defined as

$$\nabla R(\vec{y}) \equiv \neg(R(\vec{y}) \iff R'(\vec{y})) \equiv (R(\vec{y}) \wedge \neg R'(\vec{y})) \vee (\neg R(\vec{y}) \wedge R'(\vec{y}))$$

Therefore neutral axioms can now be written as $(\forall \vec{y}) \Psi(\vec{y}) \Rightarrow \neg \nabla R(\vec{y})$, or, using the contrapositive, $(\forall \vec{y}) \nabla R(\vec{y}) \Rightarrow \neg \Psi(\vec{y})$. Therefore the predicate $\neg\Psi$ can be seen to explain the conditions under which `R` may be modified as a result of the procedure execution.

In this case, the stronger the assertion Ψ, the more things remain unchanged. As before, we want to conjoin to each procedure specification the strongest possible neutral assertions.

For procedure `B$m`, the resulting frame assertions would be

$$\begin{aligned} \nabla p(\vec{y}) &\Rightarrow \vec{y} = (self, 1) \wedge \\ \nabla q(\vec{y}) &\Rightarrow false \wedge \\ \nabla r(\vec{y}) &\Rightarrow false \wedge \\ &\ldots \end{aligned}$$

while the frame assertion for `E$m` would be

$$\begin{aligned} \nabla p(\vec{y}) &\Rightarrow (\vec{y} = (self, 1) \vee \vec{y} = (self, 4)) \wedge \\ \nabla q(\vec{y}) &\Rightarrow false \wedge \\ \nabla r(\vec{y}) &\Rightarrow false \wedge \\ &\ldots \end{aligned}$$

Reverting to the set-theoretic notation of languages such as Z, we observe that ∇R corresponds to symmetric set-difference (which we denote by $\ominus$), and that the above frame assertion would therefore look like

$$p \ominus p' \subseteq \{1, 4\} \wedge q \ominus q' \subseteq \emptyset \wedge r \ominus r' \subseteq \emptyset \wedge \ldots$$

As usual, the difficulty is determining these formulas algorithmically.

[7] For simplicity, we will use the notation of predicate calculus rather than set theory here.

4.4 A model-theoretic approach

Let us look again at a problem we raised earlier, when we first noted the need for frame assertions: the inability to prove that invariants are maintained by the `Move` procedure

$$(age < 120) \wedge \text{POST\_}Move \;\; \Rightarrow \;\; (age' < 120)$$

where POST_Move was $address' = newAddress?$. As we said at the time, the problem was that the post-condition did not rule out lots of other changes; i.e., among the models of the formula $address' = newAddress?$ we find some in which age' and age differ, as well as ones where they do not. Model theoretically, we would therefore like to select out those models of POST_Move in which things change only if they had too; in other words, differences between the primed and unprimed predicates are minimized, in the sense that there is no model of the post-condition which has fewer changes.

For this, we can turn to the technique of circumscription [19], which has been used to model non-monotonicity. Circumscription essentially selects only those models of a theory which minimize the extents of some predicate(s). Circumscription may be illustrated by showing how it captures the notion of "negation by failure" present in Horn-logic based languages such as Prolog: given the assertion `P(1)` and clause `Q(x) :- P(x)`, negation by failure would allow one to conclude the following list of atoms $Q(1), \neg Q(2), \neg Q(3), ...$, while classical logic only entails the first one. This is achieved by making all predicates, including P and Q, have *the smallest extent necessary* to satisfy the originally given facts $\{P(1), P(x) \;\; \Rightarrow \;\; Q(x)\}$. [8]

In our case, we have stated that we are interested in minimizing changes, so let us introduce again the predicates ∇R. Remember the (defining) property of ∇R:

$$\nabla R(\vec{y}) \iff (R(\vec{y}) \wedge \neg R'(\vec{y})) \vee (\neg R(\vec{y}) \wedge R'(\vec{y}))$$

Technically, we are then interested in circumscribing (minimizing) the predicates in the set $\{\nabla R\}$ subject to the theory consisting of the post-condition and the above axioms defining the ∇R predicates. The minimization is to be done while keeping the extents of unprimed predicates fixed but varying the extents of the primed predicates.

For example, for the case of procedure `E$m`, with parameter `self`, we are circumscribing the predicates { ∇p, ∇q, ∇r,...} with respect to the theory consisting of the following axioms

$$p'(self, 1) \wedge p'(self, 4)$$
$$\nabla p(\vec{y}) \iff (p(\vec{y}) \wedge \neg p'(\vec{y})) \vee (\neg p(\vec{y}) \wedge p'(\vec{y}))$$
$$\nabla q(\vec{y}) \iff (q(\vec{y}) \wedge \neg q'(\vec{y})) \vee (\neg q(\vec{y}) \wedge q'(\vec{y}))$$
$$...$$

The resulting set of models is then the one in which we wish to carry out proofs concerning the maintenance of integrity constraints for example. This set of models can in fact be characterized using a second-order axiom schema (see [19]), which we shall not present here, and which would constitute the frame assertion.

This model-theoretic way of describing the frame assumption built into our specification language has the desirable property that it is self-consistent (that is the big advantage of model theories in general) and is independent of the syntax of the assertions used in the goal condition. In particular it is not affected by the introduction of tautologies, etc.

Its disadvantage (in addition to it not being clear how to compute with) is that, if applied globally, every specification requires us to minimize change so in fact there is no way to indicate

[8] We point out that although this model-theoretic description seems impossible to implement, it in fact has an effective implementation in the case of pure Horn logic — the technique of negation by failure.

that the implementer has the freedom to change or not some value (as in the case of a searching program that might want to modify the value in the array right after the last element of the list to be searched). For this purpose, we would want to introduce into the specification language some way to state that at certain points minimization is not desired.

4.5 A connection between syntactic and model-theoretic techniques

One of the great achievements of 20th century mathematics is the connection in logic between model theory/semantics and proof theory/syntax, which allows us to demonstrate the soundness and completeness of our reasoning techniques. The techniques introduced in the preceding three sections open up the exciting possibility of finding a similar relationship in the specification of frame assertions.

The following proposition, presented here without proof in the set-theoretic notation of languages such as Z, indicates that in some cases there is in fact such a connection:

> *Proposition* Suppose M is a procedure possibly modifying set-valued variables $s_1, s_2, ...$, whose post-condition is expressed as the conjunction of predications of the form $s_i' \cap \alpha_i = \beta_i$, where α_i and β_i are sets defined without using primed variables. Then the preferred models according to the circumscriptive theory in section 4.4 are exactly those that satisfy the assertions
>
> $$s_i \ominus s_i' \subseteq \alpha_i \cup \beta_i$$
>
> If some variable z does not appear in the post-condition then the assertion is simply $z \ominus z' = \emptyset$.

In other words, the circumscriptive frame axioms for M are expressed by the conjunction of the above formulas, which have the form prescribed in section 4.3.

Observe that the form of the assertions allowed in the post-condition is sufficiently rich to describe the addition or removal of (sets of) elements and conditional specifications:

- to state that $1 \in p'$, write $p' \cap \{1\} = \{1\}$
- to state that $7 \notin q'$, write $q' \cap \{7\} = \emptyset$
- to state that 1 is inserted into p if w is less than 3 ($w < 3 \Rightarrow 1 \in p'$), write

$$p' \cap \{1\} = \{\delta \mid \delta = 1, \delta \neq \delta \ if \ \neg(w < 3)\}$$

5 Summary and conclusions

We have exhibited a variety of examples arguing that in object-oriented systems a more specialized class often needs to do more than just add new local variables and new methods operating on them. These arguments are not novel — corresponding features have in fact been implemented in existing object-oriented programming languages, such as Eiffel and Taxis. And note that these languages support "principled" specialization, where (in contrast to languages such as Smalltalk, where inheritance is just a syntactic code sharing technique), a specialized method cannot just blindly over-ride what it inherits.

Some of these features do not however appear in formal specification techniques that follow the object-oriented paradigm (e.g., [26, 12, 16, 17]). For example, they do not allow the effects of a procedure to include modifications of variables in other objects, reachable from this one. A sign of this is that the semantics of such specification languages is often presented as the history of states of single objects, while global changes would require a semantics involving the simultaneous states of all objects, so that co-ordination between them would be expressible.

Secondly, we showed that these techniques run into difficulties with the need to express explicitly that some variables remain unchanged. The problem was that such frame axioms contradict any further changes introduced through specialization.

We then examined two families of approaches to the problem. One of them was based on the idea that we need to treat frame axioms differently than the parts of the post-condition describing the desired changes to be implemented by the procedure. In this treatment the specifier is usually provided with special notation to express frame axioms. With one exception, such approaches cannot deal properly with the notion that a specialized procedure need only have a stronger post-condition (which allows additional modifications to composite variables already modified).

To deal with this second problem, and in order to achieve considerable simplification in general formal specifications (not just object-oriented ones — see [8]), we examined a second approach: namely, building into the language semantics that things remain unchanged as much as possible. This approach was pioneered in formal methods by Schuman and Pitt, but is widely known in the area of non-monotonic reasoning in Artificial Intelligence. In this paper, we viewed this approach as one which *specifies* the frame axioms to be added to procedure post-conditions after inheritance is complete. The difficulty faced by these approaches was that of finding an effective algorithm for computing the frame axioms. We conjecture that this is achievable at least for the subclass of deterministic specifications and gave some evidence of this in a proposition that related two seemingly different approaches.

We note that there are additional problems with reasoning about large specifications, especially for information systems. If there are n state variables, then all the approaches we considered require $O(n)$ frame axioms in the final First Order theory. With k procedures, each of which has its own frame axioms, we would then need kn axioms — a very large number, especially if we plan to use some standard theorem prover. In [8], we introduce a simple technique which relies on reifying procedures in order to reduce the total size of the axioms to $O(k+n)$.

Finally, we remark that a third class of approaches to the frame problem could be based on the idea, mentioned in [23] and [8], that computer tools could be used to generate candidate frame axioms, which may then be edited by the specifier.

Acknowledgements

I am indebted to my collaborators, Raymond Reiter and John Mylopoulos, without whom I would not be working on the problems discussed in this paper. Eric Hehner, Klaus-Dieter Schewe and Don Cohen provided useful technical pointers and clarifications.

References

[1] Abrial, J. R., P. Gardiner, C. Morgan, and M. Spivey, "Abstract Machines", Part1 - Part4, 26 Rue des Plantes, Paris 75014 June 1988.

[2] Alencar, A. and J. Goguen, "OOZE: An object-oriented Z environment", *Proc. ECOOP'91*

[3] Borgida, A. "On the definition of specialization hierarchies for procedures", *Proceedings of the 7th Int. Joint Conf. on Artificial Intelligence*, Vancouver, B.C., August 1981, pp.254-256.

[4] Borgida, A., J.Mylopoulos, and H.K.T.Wong, "Generalization/Specialization as a basis for software specification", in *On Conceptual Modeling*, M.Brodie et al. (eds.), Springer Verlag, 1984, pp. 87-114.

[5] A.Borgida, "Features of Languages for the Development of Information Systems at the Conceptual Level", *IEEE Software, 2(1)*, January 1985, pp.63-73.

[6] A. Borgida, "Modeling Class Hierarchies with Contradictions", *Proc. ACM SIGMOD '88 Conference*, Chicago, May 1988.

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

[8] Borgida, A., J. Mylopoulos, and R. Reiter, ""... and nothing else changes": the frame problem in procedure specifications", Department of Computer Science, University of Toronto, submitted for publication, 1992. (An expanded version appears as Rutgers Tech Report #).

[9] Bretl, R., D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. Harold WIlliams, M. Wiliams, "The GemStone Data Management System", *Object Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky editors, ACM Press, New Yor, 1989.

[10] Cusack, E., "Inheritance in object-oriented Z", *Proc. ECOOP'91*, Springer Verlag LNCS #512, pp.167-179

[11] Dijkstra, E.W., *A discipline of programming*, Prentice Hall, 1976.

[12] Duke, D. and R. Duke, "Towards a Semantics for Object Z", *VDM and Z — Formal Methods in Software Development*, LNCS Vol. 428, pp.244-261, Springer Verlag, Berlin, 1990.

[13] *Readings in Nonmonotonic Reasoning*, M. Ginsberg (ed.), Morgan Kaufman, Palo Alto, 1990.

[14] Guttag, J., J.J. Horning, and J.W. Wing, "Larch in five easy pieces", TR 5, DEC Systems Research Center, 1985.

[15] Hall, A. "Using Z as a specification calculus for object-oriented systems", *VDM and Z — Formal Methods in Software Development*, LNCS Vol. 428, pp.290-317, Springer Verlag, Berlin, 1990.

[16] Lano, K., and H. Haughton, "Reasoning and refinement in object-oriented specification languages", *Proc. ECOOP'92*, Springer Verlag LNCS #615, 1992.

[17] Marshall, L., and L. Simon, "Using VDM within an object-oriented framework", *VDM'91: Formal Software Development Methods*, Springer Verlag LNCS # 551, 1991.

[18] McCarthy, J., and P. Hayes, "Some philosophical problems from the standpoint of aritificial intelligence", *Machine Intelligence 4*, pp.4463-502 (eds Melzter, B. and Michie, D.). Edinburgh: Edinburgh University Press, 1969.

[19] McCarthy, J., "Circumscription — a form of non-monotonic reasoning", *Artificial Intelligence 13*, pp.27-39, 1980.

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

[21] Mylopoulos, J., P. Bernstein, and H. Wong, "A language facility for designing data-intensive applications", *ACM TODS 5(2)*, June 1980.

[22] O.Deux et al., "The storyof O2", *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[23] Penny, D., R.C.Holt, M.Godfrey, "Formal specification in metamorphic programming", *VDM'91: Formal Software Development Methods*, Springer Verlag LNCS # 551, 1991.

[24] Reiter, R. "A Logic for Default Reasoning", *Artificial Intelligence*, 13(1):81–132, 1980.

[25] Schewe, K-D., Schmidt, J., Wetzel, I., "Specification and refinement in an integrated database application environment", *VDM'91: Formal Software Development Methods*, Springer Verlag LNCS # 551, 1991.

[26] Schuman, S.A. and D.H.Pitt, "Object-oriented subsystem specification", *Program Specification and Transformation*, L.G.L.T.Meertens (ed), North Holland, 1987, pp.313-341.

[27] Sethi, R. *Programming languages: concepts and constructs*, Addison Wesley, 1989.

[28] Spivey, J.M., *The Z notation: a reference manual*, Prentice Hall, 1989.

[29] Zave, P. and M. Jackson, "Conjunction as composition", manuscript submitted for publication, AT&T Bell Laboratories, 1992.