The Logic of Propagation in The Inscape Environment

Dewayne E. Perry
Computing Systems Research Laboratory
AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract

The Inscape Environment research project addresses issues in supporting the development of large systems by large numbers of programmers. One aspect of this research is the ‘‘constructive use’’ of formal module interface specifications - that is, given that you have formal specifications, what can you do with them. In Inscape, the specifications form the basis for providing an environment that is knowledgeable about the process of developing and evolving software systems, an environment that works in symbiosis with the programmer to develop and evolve a software system.

In this discussion, I present how Inscape uses operation specifications (based on Hoare’s input/output predicate approach) as the basis for synthesizing the interfaces for such complex languages statements as sequence, selection and iteration. In each of these statements, the synthesized interface is a function of the component interfaces.

I first present the basic rules for interface specification use and the logical framework for interface propagation and error detection. I then define the rules for propagating the interfaces for sequence, selection, iteration and operation. Finally, I define notions of ‘‘implementation completeness and correctness’’.

1. Introduction

The Inscape Environment [1] is a research project that is centered around two orthogonal concepts:

- the *constructive use* of module interface specifications, and
- environmentally supported *crowd control*.

Inscape is an experiment in developing a *city model* [2] software development environment that is based on the practical use of formal specifications in the context of large software system development by large groups of people. In this context, I am concerned about three major problems [3]: complexity, evolution, and scale. Complexity is one of the fundamental problems discussed by Brooks in his ‘‘No Silver Bullets ...’’ paper [4]. It manifests itself in three different ways: in the intricacy of detail, in the wealth of detail, and in the invisibility of both the intricacy and the wealth of detail. It is this last aspect of complexity, invisibility, that is a major focus of my Inscape research — that is, making the invisible and implicit details visible and explicit. Evolution is an extremely important aspect of software systems that has not received its proper consideration. It is not caused by the fact the ‘‘we have not got it right the first time’’; it is a basic, fundamental problem in long-lived software systems. Scale is another important problem that is underconsidered as far as supporting tools and environments are concerned — the state of the art in SDEs with respect to the problems of scale is embodied in *individual* and *family model* SDEs.

One of the main directions of research in the Inscape project is that of making practical use of specification and verification technology — that is, providing a way of constructively using specifications to assist in managing the problems of complexity, evolution and scale. This practicality is accomplished in several ways:

- by using formal module interface specifications in the construction of software systems,
- by using a *semantic interconnection model* [5] in which unit, syntactic and semantic dependencies are determined and maintained, and
- by making practical trade-offs in the areas of logic and analysis (for example, how weak a logic can be used for specifications, or how strong can incremental consistency checking be made and still be used in an interactive environment?).

The focus of this paper is on one aspect of Inscape’s logic and analysis: the logic of propagation, the synthesis of interfaces, and the determination of semantic errors in using interfaces.

In Section 2, I discuss Inscape’s interface specification language, Instress, and discuss various aspects of incremental analysis. Issues of incremental analysis within Inscape are discussed in Section 3. In Section 4, I present the logic of propagation, first providing some motivation and intuition. Then, in Section 5, I define the rules for propagation within a sequence (that is, enforcing the consistent use of interfaces and determining semantic errors), and the rules for synthesizing the interfaces for sequences, operations, selection, and iteration. I define notions of completeness and correctness in Section 6 and summarize my results in Section 7.

2. The Constructive Use of Interface Specifications

Instress is the module interface specification language for Inscape and consists of the following components:

- the specification logic (SL)[1] in which predicates are defined

that provide the basic, formal vocabulary for describing interfaces,

- C syntax for declaring interface objects (for example, types, shared data, and operations),
- SL annotations for defining the properties of data objects and the interface behavior for operations, and
- pragmatic information to assist the user of the interface.

The specification of interfaces is based on Hoare's input/output predicates [6]. I extend Hoare's approach in two ways: by adding the notion of obligations as a part of the result specification and by providing the specification of multiple results.

Hoare [7] states ''if the assumptions are falsified, the product may break, and its subsequent (but not its previous) behavior may be wholly arbitrary. Even if it seems to work for a while, it is completely worthless, unreliable, and even dangerous.'' There are indeed some assumptions whose falsification leads to arbitrary, unpredictable behavior. However, in robust, fault-tolerant software, there is a class of assumptions where wholly predicatable results are produced when the assumptions are found to be falsified: those whose failures result in exceptions [8]. For this reason, we provide the explicit specification of exceptional results in addition to normal results.

Side-effects are a basic fact of programming in Algol-like languages. Hoare's approach provides a good vehicle for capturing most of the side-effects that occur. However, there are some subsequent actions that are entailed (as side-effects) as the result of executing a piece of software that are not describable as postconditions. As an analogy, consider obtaining a loan from a bank: the postcondition is that you have the money in hand; but, you also have an obligation to pay the money back as a result of obtaining the loan. Similarly, opening a file, or allocating a buffer, have side-effects beyond the facts that the file is opened, or that the buffer is allocated. An obligation exists to close the file, or to deallocate the buffer.

Thus, Inscape provides the following extension of Hoare's paradigm:

```
Preconditions
        { Program }
                Postconditions, Obligations
                . . .
                Postconditions, Obligations
```

Since the mechanisms for propagation are the same for both normal and exceptional results, I will consider only the normal exit result in the remainder of this paper.[2]

3. Incremental Analysis

My primary strategy in Inscape for managing the problems of evolution and scale in software systems is that of static incremental analysis using the interface specifications.[3] The specifications provide a bootstrapping mechanism: assume that they are correct and use them; when they are found to be incorrect, change them and determine the implications of those changes.

Incrementality occurs at two different levels: the gross grain of increment is the operation (that is, a function or procedure, the collection of which together with data objects comprise a module); the fine grain of increment is the language statement.

By making a function or procedure the gross-grain increment, we eliminate the need to consider a system as a whole. Assuming that interfaces are correct and enforcing their consistent use provides one-half of the equation that eliminates this need for analyzing the entire system. The propagation, or synthesis, of the interfaces from the implementations of operations provides the other half of the equation.

The fine-grained increment in Inscape is the individual language statement. For complex statements such as sequence, selection and iteration, Inscape synthesizes the interface from their component parts. Describing this synthesis is the main purpose of this paper. The goal has been to generate the interfaces for these statements as independently of their context as possible.[4] In this way, changes that result in a change to a statement interface can be handled independently and propagated only to the appropriate context.

To show how this is possible, consider the following intuitive ''picture'' of how preconditions and postconditions interact: postconditions ''sink down'' through the implementation and represent the current state of the computation; preconditions ''float up'' through the implementation ''looking'' for satisfaction by postconditions. At the point where a precondition P occurs (for example as the precondition in an operation that is called in the implementation), either P is known to be true or false, or it is not known whether it is true or false.

- If P is true, then the precondition is satisfied and does not need to ''float up'' looking for satisfaction. Further, where P becomes true as a postcondition is not relevant either, at least as far as whether it is true or false.
- If P is false, then the precondition is not satisfied, but neither can it ''float up'' looking for satisfaction. It has hit a logical barrier, a precondition ceiling. Thus, where in the preceding context P becomes false is not relevant either.
- If it is not known whether P is true or false (that is, there is no reference to P whatsoever), then it is unknown in all of the preceding context as well, so that one does not need to consider the interior of preceding contexts at all (that is, the components of sequences, iterations or selections).

Thus, we can treat the implementation of an operation, a sequence, a selection, or an iteration as a black box. We need only to consider the interface of the individual statement.[5]

---

1. The form of SL is one of my current investigations.
2. For a more complete treatment of these specification issues see my paper ''The Inscape Environment'' [1].
3. Anna [9], the annotation language for Ada, on the other hand, provides dynamic analysis of annotations embedded in both interfaces and implementations. Inscape is concerned only with the static analysis of interface specifications and the propagation of those interface elements throughout the implementation.
4. We will see below where context is important, but that is where component parts of a complex language statement are concerned, not where individual statements are concerned.
5. Unfortunately, obligations are not quite so tractable. As obligations ''sink down'' looking for satisfaction, the relationship that holds between preconditions and postconditions does not hold. Thus, we have to introduce some extra mechanism for obligations. For this paper, however, we take a somewhat simplified view to show the general picture.

4. Inscape's Propagation Logic

I have separated the notions of *consistency* and *propagation* to clarify their different concerns. Consistency checking is one of the areas where trade-offs must be made in order to make the environment a practical one. The trade-offs there depend on how to manipulate the specification logic (SL). Propagation of the interfaces, on the other hand, depends on trade-offs about how to statically represent the expanding possible-execution tree as a converging, directed graph — that is, how to maintain a *single thread of knowledge*.[6] How this is done will be described in the next section.

In trying to describe the results of joining two paths of execution together (as one must do, for example, in treating an IF statement as an indivisible unit), one is tempted to use the logical **or**. For example, if P[7] and Q are true in one path, and P and R are true in the other, one is tempted to describe the results of joining the two paths as P **and** (Q **or** R). However, if later in the execution path **not** Q becomes true, the logically valid inference of R from (Q **or** R) does not really hold, because in one of the two possible execution paths, only Q would have been true and the only thing that we would know as a result is that Q is no longer true. We would know nothing about R. For this reason, we avoid the use of **or** and instead use the notion of a **possible** predicate (that is, a predicate that is possibly true or false rather than known or unknown to be true or false).

Remember that we have separated the notion of consistency from that of propagation. While the specification logic (SL) is a form of predicate calculus,[8] the predicative structure of its sentences is ignored (in general) at the level of propagation. I encapsulate the cases where the structure of the sentences in SL are important and consider those inferences to be in the domain of the specification logic (SL) and not in the domain of the propagation logic (PL). The two operations that are necessary from SL are:

consistent P is consistent with Q

isknown P is known in Q

The determination that one sentence is consistent or inconsistent with another is dependent on the consistency checking that is performed using SL, as is the determination that one sentence is known in (or can be derived from) a set of sentences. For the purposes of propagation, we consider these two operations to be primitive.

Inscape's propagation logic (PL), then, is a propositional calculus in which:

- a proposition P is either **true** (P) or **false** (¬P)
- the state of a proposition at an arbitrary point in a particular thread of knowledge is either

  unknown P is not known to be either true or false in any execution path prior to that point

  known P is known to be either true or false in all execution paths joined just prior to that point

  possible P is known to be either true or false in at least one execution path joined just prior to that point

- there is one sentence connective, **and** (usually represented by commas)
- there are the inference rules

  $+_{seq}$ a sequential addition or join based on the current state of the propositions

  $+_{par}$ a parallel addition or join based on the current state of the propositions

  $+_{con}$ a sequential addition or join based on the consistency of the propositions (that is, based on SL's notion of consistency)

Intuitively, the sequential addition rule embodies the notion of temporal sequence in which the more recent knowledge replaces that which was known earlier in the sequence — that is, in P1 $+_{seq}$ P2

- whatever is *known* in P2 supplants what is *known* in P1;
- whatever in P1 is *unknown* in P2 retains its state from P1;
- what is *possible* in P2 remains so in the result, except where it is *known* in P1 (and is thus *known* in the result);
- however, what is *possible* in P2 may reduce what is *known* in P1 to *possible* in the result.

Thus, P1 $+_{seq}$ P2 is defined as follows (''*'' represents ''unknown'', ''known'' and ''possible''; ''kno'' represents ''known'', ''unk'' represents ''unknown'' and ''pos'' represents ''possible''):

P1	P2	Result
* P	kno P	kno P
* P	kno ¬P	kno ¬P
* P	unk P, unk ¬P	*P
kno P	pos P	kno P
kno P	pos P, pos ¬P	pos P, pos ¬P
kno P	pos ¬P	pos P, pos ¬P
pos P	pos ¬P	pos P, pos ¬P
pos P	pos P	pos P
unk P	pos P	pos P
unk P	pos ¬P	pos ¬P

Note that P $+_{seq}$ Q is not symmetric.

Intuitively, the parallel addition rule embodies the notion of joining results of two independent execution paths into one result.

- Only what is *known* (*unknown*) to be true or false in both parts is *known* (*unknown*) to be true or false in a parallel join of the two parts.
- What is *known* in only one part becomes *possible* in the result.

P1 $+_{par}$ P2 is symmetric and is defined as follows

6. Actually, you have multiple threads of knowledge that eventually converge, rather than expand, in the same way that the textual form of the program does. Each result, in a general way, represents a single thread of knowledge.
7. Throughout the paper, I use the propositional symbols P, Q, . . . to represent sentences. For more explicit examples of what these sentences might be, see the example in [1].
8. Probably with limited quantification. The exact form of SL is currently being determined.

P1	P2	Result
kno P (¬P)	kno P (¬P)	kno P (¬P)
kno P (¬P)	unk P (¬P)	pos P (¬P)
kno P (¬P)	pos P (¬P)	pos P (¬P)
kno P	kno ¬P	pos P, pos ¬P
kno P	pos ¬P	pos P, pos ¬P
unk P, (¬P)	unk P, (¬P)	unk P (¬P)
unk P (¬P)	pos P (¬P)	pos P (¬P)
pos P	pos ¬P	pos P, pos ¬P

Intuitively, the consistent addition rule embodies the notion of adding the consistent portion of one set to that of another. Note that this operation is not symmetric.

- $P1 +_{con} P2$ is defined as

  $\{ p1 ... pk \in P1 \mid p1 ... pk \text{ are consistent with } P2 \} \cup P2$

  that is, the result of $+_{con}$ is P2 plus those propositions in P1 that are consistent[9] with P2.

On the basis of consistent addition, we introduce a derived rule that intuitively provides the elements in one result that are inconsistent with those of the other.

- $P1 -_{con} P2$ is defined as

  $P1 - (P1 +_{con} P2)$

  that is, the result of $-_{con}$ are those propositions in P1 that are inconsistent with P2.

5. Propagation in Inscape

The basis for propagation in Inscape is the instantiation of the interface specification of an operation by the simultaneous substitution of the arguments for the formal parameters in that specification. This instantiation is performed by the environment for function and procedure invocations.

The global invariant for the propagation of predicates is as follows:

> **every precondition and obligation is either satisfied within the implementation component or propagated to the interface of that component.**

The occurrence of preconditions or obligations that are not satisfied and have not been propagated to the interface indicate that the implementation has discernable errors.

In the subsequent discussion, we use the inference rules defined for SL, PL, and the standard set theory operations, $\cup$, $\cap$, $\subset$, $\subseteq$, $=$, and $-$. I first introduce a number of sets that are basic in reasoning about sequences, discuss the rules of propagation from one sequent to the next, and define how the interface of a sequence is synthesized from the sequence of sequents. I next define how the interface of an operation is synthesized from its implementation sequence. Given the basis of the sequence, I then show how Inscape constructs the interface for selection (IF) and iteration (WHILE and REPEAT) as functions of their component parts.

9. While the notion of consistency is part of SL and not PL, I note here that the distinction between ''possible'' or ''known'' is not part of SL but PL. As such, all possible predicates are treated as known in determining consistency. Thus, if it is the case that P is in P1 and possibly ¬P is in P2, the P will not be in the result of $P1 +_{con} P2$.

5.1 Sequences

The sequence is the basic construction unit in building software. It is in the context of sequence that I want to treat each sequent (that is, each operation invocation, if statement, etc.) as an independent, indivisible unit. It is in this context that the basic notions of precondition and obligation satisfaction, propagation and logical barriers are defined. It is in this context that the accumulation of the current state of the computation is also defined.

- The following sets are used for reasoning about each Sequent S_i in the Sequence $S = S_1 .. S_n$

Pre_i	the set of preconditions for $sequent_i$
$Post_i$	the set of postconditions for $sequent_i$
Obl_i	the set of obligations for $sequent_i$
$PreCeil_i$	the preconditions ceilinged by $sequent_i$ (that is, $sequent_i$ forms a precondition barrier)
$OblFloor_i$	the obligations floored by $sequent_i$ (that is, $sequent_i$ forms an obligation barrier)
$State_i$	the current state after $sequent_i$
$Promised_i$	the set of obligations outstanding after $sequent_i$
$Needed_i$	the accumulated set of unsatisfied preconditions up to and including $sequent_i$ (from $sequent_n$ — remember, preconditions ''float up'')
$SatPre_i$	the satisfied preconditions for $sequent_i$
$UnsatPre_i$	the unsatisfied preconditions for $sequent_i$
$SatObl_i$	the satisfied obligations for $sequent_i$

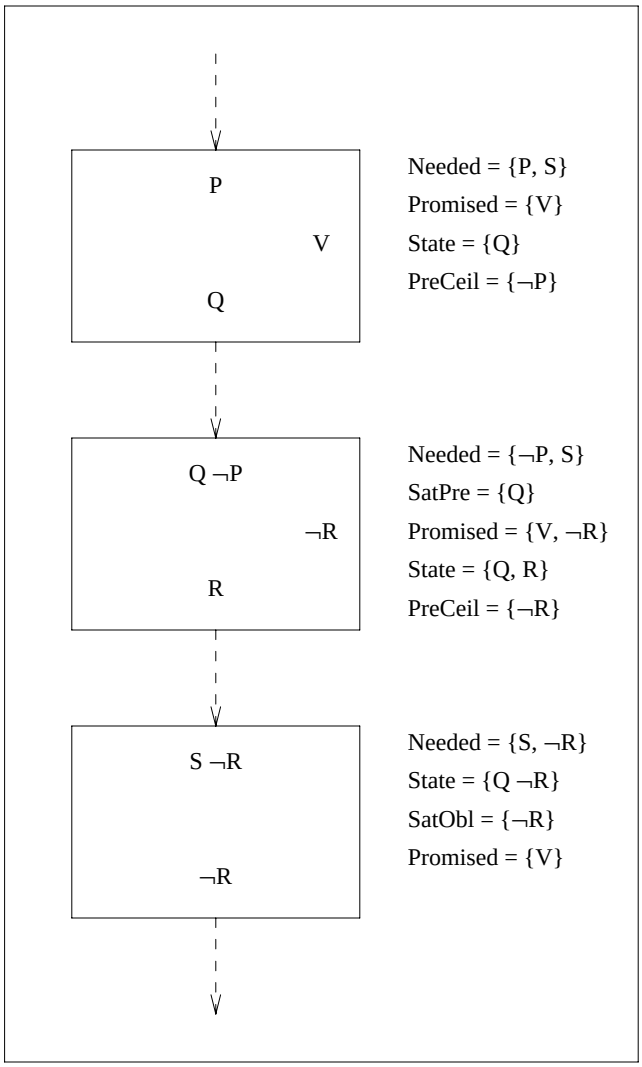
- For the Sequence S there is an initial $State_0$ and $Promised_0$
- For each Sequent S_i in Sequence $S = S_1 .. S_n$

$State_i$	$= State_{i-1} +_{seq} Post_i$
$SatPre_i$	$= \{ P \in Pre_i \mid P \text{ is known in } State_{i-1} \}$
$UnsatPre_i$	$= Pre_i - SatPre_i$
$Needed_i$	$= (Needed_{i+1} - PreCeil_i) \cup UnsatPre_i$
$PreCeil_i$	$= (Needed_{i+1} -_{con} Post_i) \cup (Needed_{i+1} -_{con} Pre_i)$
$SatObl_i$	$= \{ O \in Promised_i \mid O \text{ is known in } State_i \}$
$Promised_i$	$= ((Promised_{i-1} - OblFloor_i) \cup Obl_i) - SatObl_i$
$OblFloor_i$	$= Promised_{i-1} -_{con} Obl_i$

The current state is the result of sequentially adding the current postconditions to the previous state. The satisfied preconditions are those that are known (according to SL) in the preceding sequent's resulting state. The currently needed preconditions are those subsequently needed, less any preconditions that have been ceilinged by the current postconditions or preconditions, together with those preconditions that have not been satisfied by the previous state. The satisfied obligations are those previously unsatisfied obligations that have accumulated from the preceding and current sequents that are known in the current sequent's resulting state. Notice that the accumulated obligations do not include those obligations that have been previously or currently floored.

Precondition ceilings and obligation floors are particularly important because they represent preconditions and obligations that have failed to satisfy the global invariant state above: they have not been satisfied and have not been propagated to the

interface. They represent detected interface-use errors in the implementation.



Example 1: A Sequence

Consider Example 1 showing a sequence of three sequents[10]. The set *State* represents the sequential state of the computation, accumulating a consistent set of postconditions. The set *Needed* represents the accumulation of preconditions that have not been satisfied. The set *Promised* contains those obligations that have not yet been satisfied. Notice that there are two cases where a precondition has not been satisfied and cannot be propagated to the interface. These cases are exemplified by the non-empty *PreCeil* sets (as, as such, indicate errors in the implementation of the sequence).

The rules of synthesizing the interface for a sequence are as follows:

- Let the Sequence S = S_1 .. S_n where $State_0$ and $Promised_0$ have been initialized according to the context of the sequence.
- The interface for S is propagated as follows:

  S.Pre = $Needed_1$
  S.Post = $State_n$
  S.Obl = $Promised_n$

- The contents of S.PreCeil and S.OblFloor may be amended according to the context of the use of the sequence S.

According to these rules, the interface for the sequence in the preceding example is:

S.Pre = {P, S}
S.Post = {Q, ¬R}
S.Obl = {V}

5.2 Operations

A function or procedure consists of a set of local declarations and an implementation sequence. The rules are simple and straightforward for synthesizing the operation interface from the implementation sequence interface: remove all those predicates that refer to local data objects.

- Let the Operation O have an implementation sequence S = S_1 .. S_n where S.$State_0$ = ∅ and S.$Promised_0$ = ∅
- The interface for O is propagated as follows:

  O.Pre = S.Pre – { P | P refers to local variables }
  O.Post = S.Post – { P | P refers to local variables }
  O.Obl = S.Obl – { P | P refers to local variables }

- The state of the sequence S is amended as follows:

  S.PreCeil = S.Pre – O.Pre
  S.OblFloor = S.Obl – O.Obl

Thus, any preconditions or obligations that cannot be propagated because they refer to local objects appear in the precondition ceilings and obligation floors of the implementation sequence. If in the preceding example, S, Q, and V refer to local variables, then the interface for the operation for which this sequence is the implementation sequence would have the following interface:

O.Pre = {P}
O.Post = {¬R}
O.Obl = { }

Further, the implementation sequence would now have the following non-empty sets:

S.PreCeil = {S}
S.OblFloor = {V}

5.3 Selection

I present the IF statement as the representative example of selection. The CASE statement is easily generalized from the IF statement.

- Let the Selection Statement S consist of BE = the boolean expression, T = the then sequence, and E = the else sequence. The boolean expression, BE, has two results (true and false) each of which consist of a set of postconditions and a set of obligations (BE.True.Post and BE.True.Obl, and BE.False.Post and BE.False.obl).
- The then and else sequences, T and E, are initialized as follows:

10. In this example, the pictorial syntax is a follows: preconditions appear at the top center of each box, obligations at the right center of each box (if there are any), and postconditions appear at the bottom center of each box.

$T.State_0$ = BE.True.Post
$E.State_0$ = BE.False.Post
$T.Promised_0$ = BE.True.Obl
$E.Promised_0$ = BE.False.Obl

- The interface for S is propagated as follows:

  S.Pre = BE.Pre ∪ (T.Pre – T.PreCeil) ∪ (E.Pre – E.PreCeil)
  S.Post = T.Post + E.Post
  S.Obl = T.Obl ∩ E.Obl

  The propagated preconditions are those which are required independent of which path is to be traversed (less those preconditions that are ceilinged in the then and else sequences). The propagated postconditions are the result of adding in parallel the results of the then and else sequences (resulting in known and possible predicates). Only those obligations that have been entailed in both sequences are propagated.

- The state of the selection statement S is amended as follows:

  T.OblFloor = T.Obl – S.Obl
  E.OblFloor = E.Obl – S.Obl
  T.PreCeil = (T.Pre – BE.True.Post) ∪ (T.Pre – E.Pre) ∪ (T.Pre – BE.Pre)
  E.PreCeil = (E.Pre – BE.False.Post) ∪ (E.Pre – T.Pre) ∪ (E.Pre – BE.Pre)

  Obligation floors contain those obligations that have not been propagated. Precondition ceilings contain those preconditions which are inconsistent with the postconditions and preconditions or the boolean expression, and inconsistent with the other sequences propagated preconditions.[11]

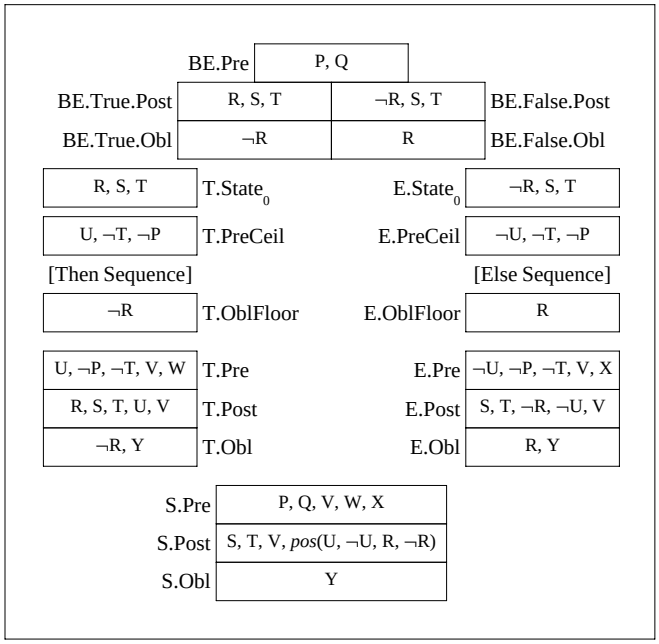
Consider Example 2, which illustrates an IF statement. The initial state of the then and else sequences are set from the true and false results respectively. The interface for the IF statement is generated as follows: the preconditions P and Q are the boolean expression preconditions, while V, W, and X are the preconditions from the then and else sequences that are consistent with each other, with the postconditions of their respective boolean expression results, and with the boolean expression preconditions; the postconditions S, T, and V are those postconditions that are known independent of which sequence was executed, while U, ¬U, R, ¬R are those postconditions that are known in only one of the two sequences (and hence are only possible in the result); the obligation Y is that obligation that is entailed independently of which sequence was executed.

Preconditions U and ¬U are ceilinged because they conflict with each other; ¬T is ceilinged because it conflicts with the boolean expression postconditions T; and ¬P is ceilinged because it conflicts with the boolean expression precondition P.

Obligations R and ¬R are floored because they do not occur in both sequences.

5.4 Iteration

Iteration is both simpler and more complex that selection. It is simpler because it has only one sequence as a part of it. It is more complex because that sequence loops back on itself — that is, the results of the loop body sequence can have an effect on the preconditions of the statement, and, worse, can cause an error by being inconsistent with the boolean expression preconditions. Note in the rules below that there is no notion of invariant, nor is there any consideration of the results of the loop body beyond the one error condition and their propagation to the interface as ‘‘possible’’. Detecting the error of the loop body postconditions being inconsistent with the boolean expression is all that is needed for the following reasons: if nothing is known in the loop body about a particular boolean expression precondition P, then whatever postcondition outside the loop that satisfies it will remain known throughout the loop body; whatever is known or possible in the loop body, will either satisfy that precondition P, or conflict with it — we only need to worry about the conflict.



Example 2: Selection Statement

The rules for WHILE are defined as follows:

- Let the Iteration Statement W consist of BE = the boolean expression, and B = the loop body (a sequence). The boolean expression, BE, has two results (true and false) each of which consist of a set of postconditions and a set of obligations (BE.True.Post and BE.True.Obl, and BE.False.Post and BE.False.obl).

- The loop body, B, is initialized as follows:

  $B.State_0$ = BE.True.Post
  $B.Promised_0$ = BE.True.Obl

- The interface for W is propagated as follows:

  W.Pre = BE.Pre ∪ (B.Pre – B.PreCeil)
  W.Post = (B.Post + ∅) + BE.False.Post
  W.Obl = BE.False.Obl

  Note that the loop body postconditions are changed from known to possible (since indeed it is possible to skip the body entirely if the boolean expression is false) by applying the parallel addition to the postconditions and the empty set.

- The state of the loop body sequence B is amended as follows:

11. Note that I have taken a rather stronger position on the flooring of obligations than might be taken. One could propagate them conditionally instead. However, ‘‘conditional propagation’’ is considerably more difficult and we have taken the simpler approach.

$$\begin{aligned} B.OblFloor &= B.Obl \\ B.PreCeil &= (B.Pre -_{con}^{con} BE.True.Post) \cup \\ &\quad (B.Pre -_{con}^{con} BE.Pre) \cup (B.Pre -_{con} B.Post) \end{aligned}$$

Note that we prevent any loop body precondition from being propagated to the loop body interface if it conflicts with the propagated postconditions.

- There is an Error when $W.Pre -_{con} B.Post \neq \varnothing$

The REPEAT[12] statement differs slightly from the WHILE in that the results of the loop body are known, not just possible. Similarly, the obligations are treated as if the loop body were in the sequence just prior to the boolean expression (which it in fact is).

- Let the Iteration Statement R consist of BE = the boolean expression, and B = the loop body (a sequence) where

$$\begin{aligned} B.State_0 &= BE.False.Post \\ B.Promised_0 &= BE.False.Obl \end{aligned}$$

- The interface for R is propagated as follows:

$$\begin{aligned} R.Pre &= BE.Pre \cup (B.Pre - B.PreCeil) \\ R.Post &= B.Post +_{con} BE.True.Post \\ R.Obl &= B.Obl +_{seq}^{seq} BE.True.Obl \end{aligned}$$

- The state of the loop body sequence B is amended as follows:

$$\begin{aligned} B.OblFloor &= B.Obl - R.Obl \\ B.PreCeil &= (B.Pre -_{con}^{con} BE.False.Post) \cup \\ &\quad (B.Pre -_{con}^{con} BE.Pre) \cup (B.Pre -_{con} B.Post) \end{aligned}$$

- There is an Error when $R.Pre -_{con} B.Post \neq \varnothing$

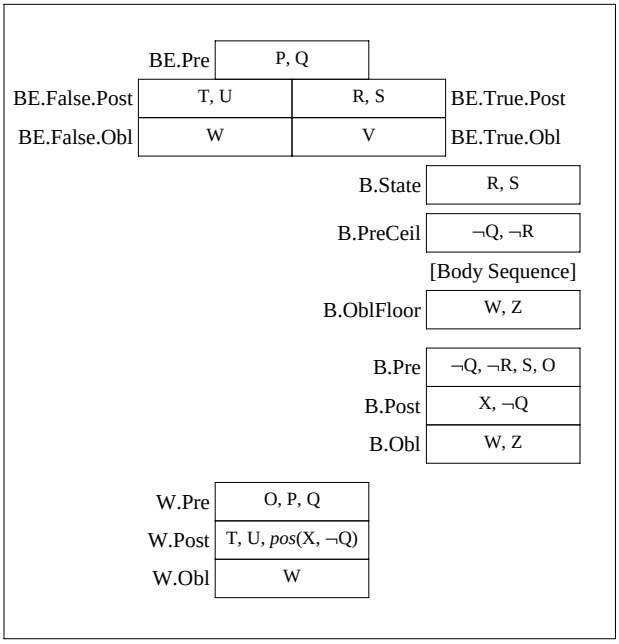
Consider Example 3, which illustrates the WHILE statement. The initial state of the loop body sequence is set from the true results thus yielding the initial state of {R, S} and the initial promised of {V}. The interface of the WHILE is generated as follows: the preconditions P and Q are the boolean expression preconditions, while O is the precondition from the body sequence that is consistent with the boolean expression true results and the boolean expression preconditions; the obligation W is that of the boolean expression false result; the postconditions T and U are those of the false result, while X is a possible postcondition from the body sequence interface. Notice that ¬Q produces an error: it conflicts with with the boolean expression precondition Q.

Precondition ¬Q is ceilinged because it conflicts with the boolean expression precondition Q; and ¬R is ceilinged because it is inconsistent with the true result postcondition R.

Obligations W and Z are floored because the loop body sequence may not be executed.

6. Completeness and Correctness

As mentioned earlier, the basic rule in Inscape is that every precondition and obligation must be either satisfied within an implementation or propagated to its interface. I have shown examples (in the operation, if and while interfaces) where this does not happen — where there exist precondition ceilings and



Example 3: Iteration Statement

obligation floors. Precondition ceilings and obligation floors contain unsatisfied predicates and, as such, indicate that there are errors. The notion of *implementation completeness* is defined to determine this kind of problem — an implementation is complete if there do not exist any detected[13] errors.

An implementation I = sequence $S = S_1 .. S_n$ for an operation O is *complete* if and only if

- Every precondition in S has either been satisfied or is in the interface of O — that is, all precondition ceilings in S (recursively) are empty
- Every obligation in S has either been satisfied or is in the interface of O — that is, all obligation floors in S (recursively) are empty.
- There are no iteration errors — that is, $I.Pre -_{con} B.Post = \varnothing$ for all iterations.

Inscape provides a slight variation of the notion of correctness of an implementation. In addition to the standard notion of correctness with respect to a specified interface, I require that the implementation be complete as well.

A propagated interface PI for operation O is *correct* with respect to the specified interface SI for operation O if and only if

- the implementation I for operation O is complete
- the interfaces PI and SI are identical
  - PI.Pre = SI.Pre
  - PI.Post = SI.Post
  - PI.Obl = SI.Obl

Note: Redefinition of the propagated interface may be needed to cast it in terms of the specified interface.

12. Remember that the form of the REPEAT statement is ‘‘**repeat** <loop-body> **until** <boolean-expression>’’. The loop body is executed at one or more times until the boolean expression is becomes true.

13. Please note that I do **not** claim that no errors exists when an implementation is complete, only that there are none that I am able to detect.

The completeness requirement is not strictly needed for the definition, but it seems somewhat incongruous to have detected errors in the implementation and still consider it correct.

7. Summary and Current State of the Project

The novelty of Inscape's approach lies in its constructive use of interface specifications. The formal interface specifications are the basis for *constructing* (or propagating, synthesizing) interfaces from component parts: the interface of a sequence as a function of the constituent sequents; the interface of an operation as a function of its local declarations and implementation sequence; the interface of selection as a function of its boolean expression and then and else sequences; and the interface of iteration as a function of its boolean expression and loop body. Moreover, Inscape uses the rules of constructing interfaces to expose semantic errors in the *use* of the interfaces.

The basic mechanisms used to define sequence, selection and iteration are used to define the formal handling of exceptions as well. I have not presented the rules for the various formalized ways of handling exceptions because of length considerations. They have, however, been defined.

The work described in this paper has been implemented in the Inscape Prototype. The rules for handling exceptions have not yet been implemented, but are next on the list for implementation.

My current work in the logical and analytical aspects of Inscape is concerned with the following:

- The Propagation Logic (PL): determine its formal properties and the proofs of the various rules (the reasoning has only been done informally).
- Obligations: complete the work on the implications of not having the same relationship that preconditions and postconditions have.
- The semantics of language statements: one of the main questions is how much can the environment do automatically with assignment, and how much interaction will be needed with the user;[14] I think that little can be automated with expressions and, thus, interaction will be required to determine the interfaces of expressions (though of course the programmer can encapsulate expressions in functions and thus eliminate interaction where expressions occur more than once).
- The Specification Logic (SL): determine its form and how consistency determination can be strengthened and made more efficient (the primary emphasis, currently, is upon pattern matching and simple deduction).
- Evolution: how to make efficient use of the semantic interconnections (established by the process of propagation) and efficiently evaluate, incrementally, the effects of changes both to interfaces and implementations.

Acknowledgements

Bill Hopkins and David Rosenblum provided careful readings and insightful comments an earlier version of this paper.

14. One of my primary strategies is to automate as much as possible and interact where automation is not possible.

References

[1] Dewayne E. Perry. ''The Inscape Environment.'' *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh PA, May 1989. pp 2-12.

[2] Dewayne E. Perry and Gail E. Kaiser. ''Models of Software Development Environments.'' *Proceedings of the 10th International Conference on Software Engineering*, Raffles City, Singapore, April 1988. pp 60-68.

[3] Dewayne E. Perry. ''Industrial Strength Software Development Environments'', *Proceedings of the IFIPS '89 World Computer Conference* San Francisco CA, August, 1989.

[4] Frederick P. Brooks, Jr. ''No Silver Bullet: Essence and Accidents of Software Engineering'', *Computer*, 20:4 (April 1987), pages 10-20.

[5] Dewayne E. Perry. ''Software Interconnection Models.'' *Proceedings of the 9th International Conference on Software Engineering*, Monterey CA, March 1987. pp 61-69.

[6] C. A. R. Hoare. ''An Axiomatic Approach to Computer Programming.'' *CACM* 12:10 (October 1969). pp 576-580, 583.

[7] C. A. R. Hoare. ''Programs are Predicates.'' In *Mathematical Logic and Programming Languages*. Prentice-Hall, 1985.

[8] J. B. Goodenough. ''Exception Handling: Issues and a Proposed Notation'', *Communications of the ACM*, 18:12 (1975). pp 683-696.

[9] David Luckham and Friedrich W. von Henke. ''An Overview of Anna, A Specification Language for Ada.'' *IEEE Software*, 2:2 (March 1985). pp 24-33.