An inversion capability for the PRESTIGE workbench: Some basic issues

B Ratcliff

Dept. of Computer Science & Applied Mathematics
Aston University, Birmingham, UK.

ABSTRACT

The PRESTIGE workbench is an integrated CASE environment intended to provide full implementation support for Jackson System Development (JSD). JSD is an operational software development method, and thus implementation in JSD is essentially a transformational process. The main objective is to offer a generalised transformational facility that the JSD implementor can apply as desired to suit the needs of any particular implementation scenario, although a default implementation capability is also provided. Many of the transformations required can be automated, and the so-called ‘inversion’ transformations play a pivotal role in that capability. This paper addresses a variety of technical issues that need to be considered in providing a user-controlled inversion facility within the PRESTIGE environment.

INTRODUCTION

The Operational Paradigm and JSD

Although currently much less fashionable than other approaches, particularly object-oriented, a number of operational development methods exist, such as JSD [1,2,3,4,5], Me-too [6], PAISley [7,8] and Gist [9,10]. The operational paradigm is discussed in [11,12]. JSD is the best known example of an operational method that is actually in use in UK commerce and industry. Its three main phases are typical of an operational approach:

Modeling: From elicited requirements, an abstract model is developed of the ‘real world’ with which the system will interact. In JSD, this is accomplished by constructing life histories of relevant entities. A life history specifies the legal sequential ordering of actions that the entity can perform or experience over its lifespan.

Specification: A system specification is developed based on the real-world model and elicited functional requirements, etc. In JSD, the main ingredient is a network architecture of ‘long-running’ asynchronous sequential processes. Some of these processes are based on the real-world model and carry out the task of encapsulating and maintaining the persistent system state; others provide the functionality required of the system.

Implementation: The specification is transformed into a product that will execute acceptably within the desired target environment(s). In JSD, this requires employing various kinds of transformation, such as inversion (basically, getting rid of unwanted concurrency), state-vector separation (creating the system’s ‘database’) and dismemberment (e.g. dividing processes into functionally separate modules).

A bonus of an operational approach is that system specifications are minimally constrained by inessential specificity and determinism, thus providing high implementation freedom. Moreover, specifications are also executable (at least in principle), and implementation transformations are correctness-preserving and automatable. The use of executable specifications in software engineering is discussed in [13]; they have been applied successfully in a number of contexts [14,15,16,17,18]. Transformation systems — reviewed in [19] — can be used to generate trial software by customised instantiation of generalised abstract specifications. Not only does this represent direct support for prototyping [23], it is also a way of promoting reusability at a higher level than code [24].

The PRESTIGE Implementor’s Workbench

The PRESTIGE (PRoduction of Executable Systems via Transformational Implementation GEneration) workbench is intended to provide full, flexible support for the implementation phase of JSD. CASE tools already exist for JSD’s first two phases and the workbench is designed to interface smoothly with these. PRESTIGE uses a two-stage implementation process model. The rationale for this approach is that it provides a useful ‘separation of transformational concerns’ between aspects that are quite independent of the target language — primary transformations (inversion, etc.) addressing mainly architectural considerations and carried out in the first stage — and secondary transformations, i.e. essentially code-level manipulations, which are carried out in the second stage. The approach also enhances toolkit generality, flexibility and portability.

The purpose of this paper is not to explore the design and architecture of the PRESTIGE environment itself (brief ‘current status’ details are given in the concluding section), but to address a central technical aspect, namely inversion, which is a category of transformation of pivotal importance to the whole of the workbench’s primary transformational capability.

AN INVERSION CAPABILITY

Specification clusters

Just as a JSD specification can be built up compositionally, process by process, function by function, and/or subsystem by subsystem, it is also desirable that implementation can be tackled in a similar incremental manner. Hence, the user must be able to select any portion of a JSD specification that s/he is currently interested in implementing. Any such selected portion of a specification is called a specification cluster (SC). Any SC selected must be ‘coherent’. Consider Figure 1, which depicts the main ingredients of JSD specification architecture (circles are ideal FIFO message streams, boxes are the long-running
asynchronous processes, and triangles are inspections of system state that persist within processes. Clearly, if the user identified F, P and G as the set of processes in an SC, this would not be feasible. Likewise, if the cluster S2 was chosen for implementation, but R had already been committed to a previous SC involving parts of S3, say, then this selection would be rendered invalid. SC selection itself is provided via the options of (sub-)network 'rubber-banding' or by component-by-component selection of the appropriate part of the network. Both techniques are desirable for the user — the first is quicker and more convenient, especially when it is required to apply a default implementation strategy to a large SC; but the second will be needed when, for example, an irregular SC topology makes rubber-banding impossible.

Inversion

Given any SC, it is possible to derive by inversion transformations a number of inversion hierarchies (IHs) that implement the SC. The two basic forms of inversion are illustrated in Figure 2. The detailed mechanics of these transformations are well documented elsewhere [1,2,3] and mainly do not concern us here. Suffice it to say that, in the case of read inversion, the writes to X in P are replaced by resumes of Q; the reads from X in Q become suspend points (except the first read, which effectively is discarded). Write inversion is the inverse of this, though not symmetrically so (the first write in P must also become a suspend point). The effect of either, however, is to map an asynchronous producer/consumer pair of processes into a subroutine hierarchy that functions as a coroutine relationship. Note that subroutine connections in an IH diagram denote directional parameter passing (which in effect replaces communication by message stream in the abstract specification).

Since the PRESTIGE workbench provides the implementor with full implementation freedom, all valid IHs must in principle be permitted. Suppose, for example, that cluster S2 of Figure 1 has been selected. If read inversion is applied everywhere, and ignoring all other considerations such as possible technical invalidities, the IH will be as in Figure 6(a) of the Appendix. There are also five other IHs possible, which are also shown in Figure 6 and are discussed later (the 't' prefix to subroutine names is merely intended to denote 'transformed'). The origin of each IH depends on:

a) which process is chosen as the superordinate routine;
b) the choice of read inversion or write inversion between each pair of processes communicating by message stream (this choice is partly constrained by (a)).
Schedulerless implementation

It is possible to invert a whole SC without introducing an extra explicit scheduling process by choosing one of the processes to

- The SC needs no buffering (see later) in its implementation;
- The SC has external input to one process only (this process becomes the superordinate process).

In practice, this means that 'schedulerless' implementation will not be possible in the majority of cases because criterion (b) in particular is only likely to hold for systems with trivial input traffic requirements.

Ignoring input B, Figure 6(a) would in fact be an example of a viable schedulerless implementation. Imagine, however, that in Figure 1 the input B to R is not combined with Y or Z. Whilst the stream B could be read directly by tR in the IH of Figure 6(a), we would have a problem — tR could get 'hung' if, when called, it wanted a B message, but the B message stream was currently empty. This potential real-time problem is the reason for stipulation (b) above. The general solution is to introduce a scheduling routine to handle the input (in the example, the scheduler would of course handle A as well as B), and read-invert the appropriate consumer processes with respect to the scheduler.

As it stands, however, not only would the hierarchy of Figure 6(a) be undesirable, it would in fact be an impossible implementation of cluster S2, given that B is combined with Y and Z. This is due to a property of so-called 'rough-merged' combinations of message streams.

Rough merges

A rough merge is where several separately sourced message streams are read as a single stream by some consumer process, such as the streams B, Y and Z in Figure 1. Since these streams appear as a single stream to R, it is not in general possible to distinguish in R's text between inputs of B and inputs of Y or Z — R can only be inverted with respect to all its reads of the complete rough merge, and not to some of its reads involving only Y and Z, say. We have the general principle:

INV1: For the purposes of inversion, a rough merge can only be treated 'holistically' — the nature of rough merges means that the reads in the to-be-inverted process cannot be discriminated as regards the individual message streams that they consume.

The solution required in this particular example is simple — a scheduler is needed to handle B (and A), and R also then gets inverted with respect to the scheduler to receive its B messages; see Figure 7(a) in the Appendix.

However, INV1 leads to a further consideration. Consider write inversion of R with respect to its rough-merged input. This is also impossible, because tR could only separately call tP and tQ, say, when it 'knew' that it wanted Y or Z messages respectively; but this would require message stream discrimination within R, which would violate INV1. In fact, as a consequence of that principle:

INV2: Rough merges can only be implemented by read inversion.

Multiple inversion

Multiple inversion occurs when a process suffers inversion with respect to two or more separate message streams that it handles. This will create a so-called channel connection if at least one read and one write stream are involved between one process and another. Channel connection itself raises some non-trivial implementation issues that lie outside the scope of this overview.

Two examples of multiple inversion are given in Figure 3. Figure 3(a) indicates how channel connection can be used to generate an alternative to a standard read-inversion IH. Figure 3(b) is in fact an impossibility. Suppose tP calls tQ with a Y message — clearly, if during this resumption, tQ produces a Z message for tR, there is a problem! (similar conceptual difficulties arise if tR calls tQ wanting a Z message). In fact:

INV3: Multiple inversion of a process can be accomplished only with respect to one other process; it is not possible to multiple-invert a process — whatever the mixture of read and write inversions involved — with respect to two or more separate processes.

The IH of Figure 3(b) could be partially retained by introducing buffering (see below) between tR and tQ. However, buffering between tP and tQ would not be possible. This is because INV3.
would still be violated; the only difference would be that \( tQ \) would now be read-inverted with respect to a scheduler routine instead of \( tP \).

**Buffering**

If implementation conditions dictate, buffering can always be introduced, even when technical considerations do not otherwise demand it. There are at least three stock situations, however, where buffering is unavoidable:

- **INV4**: Buffering is necessary when
  a) performing read inversion on separate non rough-merged streams (see below);
  b) dealing with circuits in networks;
  c) implementing an IH that would otherwise violate INV3.

As regards case (b), some (though not all) circuits of length 2 can optionally be dealt with by multiple inversion, yielding a simple channel connection; see Figure 4(a). For circuits of length \( k \geq 3 \) or more, buffering is unavoidable and the circuit needs to be 'cut' at any one of its \( k \) edges; see Figure 4(b). Note that buffering always requires introduction of an additional process to handle the buffer — that is, to inspect the buffer and, as and when necessary, obtain a message from the buffer to pass it on to the routine that needs it. Generally, that additional process is the top-level scheduler itself.

Dealing with case (c) is complicated in the general case. Basically, all invertible connections must each be cut with a separate buffer except one, which should be read-inverted if possible. If there are several read-invertible connections because of separate non rough-merged inputs, then all but one of these should be buffered, again leaving a single read-inverted connection.

**Write inversion**

Read inversion, in conjunction with other techniques such as buffering, can always be applied to deal with an arbitrary IC topology. That is, write inversion is not essential in principle, though in practice it may help to simplify the implementation of some SC topologies and avoid certain inefficiencies.

**INV5**: Only one form of inversion is strictly necessary. Read inversion is preferred since this yields input-driven hierarchies. Without this, various scheduling difficulties (such as real-time 'hang-up') can arise.

To illustrate with a typical situation, consider Figure 5. Since \( X \) and \( Y \) are not rough merged, straightforward read inversion is not possible here, since \( R \) must therefore necessarily control the consumption of the two streams. However, write inversion provides a simple solution; see Figure 5(a). Nevertheless, write inversion can be avoided if we are prepared to use buffering and introduce an extra process that handles the buffering of \( X \) and \( Y \) messages; see Figure 5(b) for one possible arrangement. Note that there is now a channel connection between the buffer handler \( BUFF \) and \( R \). This is necessary because each time \( R \) suspends, it must tell \( BUFF \) via a request \( REQ \) which type of message \( (X \) or \( Y) \) it needs next.
Consider again cluster S2 in Figure 1, together with the six derivable, though defective, IHs shown in Figures 6(a)-(f) of the Appendix. Figures 7(a)-(f) of the Appendix give corresponding amended IHs that overcome the problems in the IHs of Figure 6. We shall deal with each case one by one.

**PQR**: The input B to tR is impossible (INV1) and would need to be conveyed to tR (along with A to tP) via a higher-level scheduling routine as shown in Figure 7(a). In fact, this is the IH that would result from the following default implementation strategy, often referred to as the 'knitting needle':

- Thread all external inputs together and handle them as a single rough-merged input via a scheduler process;
- The processes consuming external inputs are hung as read-inverted subroutines directly from the scheduler;
- Read inversion is applied elsewhere to form the rest of the IH, with write inversion being used only where it leads to the avoidance of buffering by the creation of channel connections;
- Circuits are cut when present, yielding buffering (there are no circuits in the example).

**PRQ**: As well as the impossible input B, tQ violates INV3. Accordingly, the tR-tQ connection (write inversion) would need to be cut with a Z-buffer, yielding the implementation of Figure 7(b).

**QPR**: Input A is not possible, though not desirable (real-time hang-up); input B is impossible as in the PRQ case. The tQ-tP write inversion must be cut with a buffer. Figure 7(c) shows the result. Note that cutting the tQ-tR connection instead (with a Z-buffer) would not yield a valid IH, since tP would then violate INV3.

**QRP**: A is undesirable, B impossible, and tP violates INV3. To overcome the latter, both tR-tP and tQ-tP connections must be cut. This is necessary for otherwise tP would still violate INV3 (note that it will get A via read inversion with respect to the scheduler); in addition, the latter cut is necessary anyway to make Q 'hangable'. The result is Figure 7(d).

**RQP**: A is undesirable, and tQ violates INV3. Again, to avoid violation of INV3, both write-inversion connections must be cut, giving Figure 7(e).

**RQP**: A is undesirable, and tP violates INV3. All three connections must in fact be cut. Figure 7(f) shows the outcome.

Note how, in all these somewhat artificial examples, the amended outcome in each case leads to an IH involving only read inversion. This is not a co-incidence. In general, application of the various inversion principles will lead to IHs dominated by read-inversion.

What Figure 7 does not convey are optimal implementations — optimal in the sense of minimising buffering. In fact, most of the buffering shown is entirely avoidable simply by the process of applying what is essentially write inversion to the buffered communication (yielding in each case a channel connection). More specifically:

- **IH(b)**: Both buffers are avoidable; note that this yields a channel connection to tP involving three separate streams: input A, output X, output Y;
- **IH(c)**: Y-BUFF is avoidable, Z-BUFF is necessary;
- **IH(f)**: All buffering is avoidable.

There is a principle worth articulating here:

**INV6**: If a routine directly connected to a scheduler (with any inversion combination) is also buffer-connected to the scheduler, the buffering can be removed by applying write inversion to the buffered message stream.

It is left as a simple exercise for the reader to derive optimal implementations by applying this principle to the architectures of Figure 7. It should be noted that the optimised form of Figure 7(f) is an IH that is completely 'flat' — that is, all routines involved are hung directly from the scheduler. In fact:

**INV7**: It is always possible to produce a completely flat implementation, i.e. a hierarchy with a uniform depth of just one subroutine.

INV7 actually embodies another default implementation strategy, which could be loosely described as multiple-inverting each process with respect to all its inputs and outputs. This results in every routine being directly channel-connected to a scheduler routine. This simplistic strategy has the advantage that it will work with any arbitrary network.

**CONCLUSION**

A prototype workbench is currently under development on an Apple Macintosh™ using Parcplace Smalltalk™. The workbench interfaces with a specification environment comprising various tools running on a separate IBM PC compatible. As regards the present functionality of the toolkit, the main capabilities are:

- Inversion, simplified state-vector separation, and dismemberments.
- Default implementation generation via the 'knitting needle' strategy.
- Exception handling — in particular, warning the user of undesirable decisions or defending the user from attempting invalid transformations.
- A highly usable, predominantly graphical user-workbench interface.

Future work is intended to augment the existing set of transformations, enhance the database generation facility, and providing code generators for languages like Ada. A usable first-generation prototype is expected by the end of 1991.

In conclusion, there has not been enough space in this paper to explore the full spectrum of transformations required in PRESTIGE. Rather, we have concentrated on exploring a number of technical issues in providing a fully-fledged inversion facility within the workbench, and have identified several basic principles involved.

**Acknowledgements**

- This work has been carried out with the support of the Procurement Executive, Ministry of Defence.
• Thanks to project colleagues Maeve Boyle and Andrew Bass for the numerous discussions that have helped to 'get a handle on' inversion and related issues.

• John Cameron of LBMS Ltd, U.K. provided several helpful comments on much of the material of this paper when it first appeared in an earlier technical report.

REFERENCES


APPENDIX

Figure 6 All IHs (ignoring defects) derivable from cluster S2 of Figure 1.

Figure 7 Amended, non-optimal IHs derived from those of Figure 6.