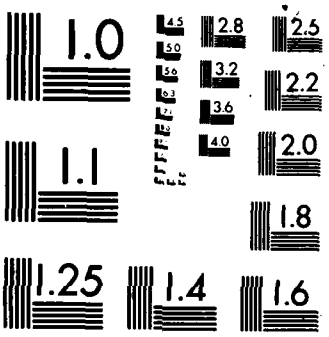
AD-A141 514 REUSE IN THE CONTEXT OF A TRANSFORMATION BASED METHODOLOGY(U) UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFORMATION SCIENCES INST M S FEATHER APR 84 1/1

UNCLASSIFIED ISI/RS-83-125 MDA903-81-C-0335 F/G 9/2 NL



ISI Reprint Series
ISI/RS-83-125
April 1984

University of Southern California

Martin S. Feather



# Reuse in the Context of a Transformation Based Methodology

Reprinted from *Proceedings of the Workshop on Reusability in Programming*, Newport, Rhode Island, 7–9 September 1983.



INFORMATION SCIENCES INSTITUTE

213/822-1511

4676 Admiralty Way/Marina del Rey/California 90292-6695

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>ISI/RS-83-125 | 2. GOVT ACCESSION NO.<br>AD-A141 514 | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE *(and Subtitle)*<br>Reuse in the Context of a Transformation Based Methodology | | 5. TYPE OF REPORT & PERIOD COVERED<br>Research Report |
| | | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s)<br>Martin S. Feather | | 8. CONTRACT OR GRANT NUMBER(s)<br>MDA903 81 C 0335 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>USC/Information Sciences Institute<br>4676 Admiralty Way<br>Marina del Rey, CA 90292-6695 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Defense Advanced Research Projects Agency<br>1400 Wilson Blvd.<br>Arlington, VA 22209 | | 12. REPORT DATE<br>April 1984<br>13. NUMBER OF PAGES<br>15 |
| 14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*<br>..... | | 15. SECURITY CLASS. *(of this report)*<br>Unclassified<br>15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT *(of this Report)*

This document is approved for public release; distribution is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

.....

18. SUPPLEMENTARY NOTES

This report is a reprint of a paper that appears in the proceedings of the Workshop on Reusability in Programming, held in Newport, Rhode Island, 7–9 September 1983. The workshop was sponsored by ITT Programming, Stratford, Connecticut.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

maintenance, program development, program specification, program transformation, reusability

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. ABSTRACT

~~Our~~ research group at ISI aims to improve the program development process by applying ***program transformation*** to develop implementations for specifications. Following this methodology, the development of a piece of software involves its specification in a formal specification language, and subsequent machine-assisted transformation of that specification into an implementation (conventional program). Subsequent maintenance and modification of software developed in this manner is achieved by modifying the specification, and reperforming the transformational development to derive a new implementation. Thus reuse occurs through reusing the original specification, and reusing the original transformational development of that specification.

~~Our~~ approach is distinguished by the nature of our specification language, which has been designed to minimize the gap between informal conceptualization and formal specification. A beneficial result of this is that maintenance and modification at the specification level is relatively straightforward. Further, the techniques that ~~we apply~~ in transforming specifications into implementations are themselves applied repeatedly, and serve to capture our programming knowledge in a conveniently reusable manner. Consideration of an example drawn from the domain of process control illustrates these points.

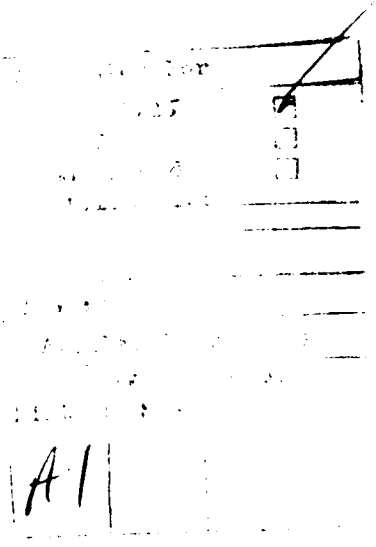
*ISI Reprint Series*
*ISI/RS-83-125*
*April 1984*

*University of Southern California*

**Martin S. Feather**

# Reuse in the Context of a Transformation Based Methodology

Reprinted from *Proceedings of the Workshop on Reusability in Programming*, Newport, Rhode Island, 7-9 September 1983.



*INFORMATION SCIENCES INSTITUTE*

*213/822-1511*

*4676 Admiralty Way/Marina del Rey/California 90292-6695*

This research is supported by the Defense Advanced Research Projects Agency under Contract No. MDA903 81 C 0335 Views and conclusions contained in this report are the author's and should not be interpreted as representing the official opinion or policy of DARPA, the U S Government, or any person or agency connected with them

*ISI Reprint Series*

This report is one in a series of reprints of articles and papers written by ISI research staff and published in professional journals and conference proceedings. For a complete list of ISI reports, write to

Document Distribution
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695
USA

# REUSE IN THE CONTEXT OF A TRANSFORMATION BASED METHODOLOGY

**MARTIN S. FEATHER**

USC / Information Sciences Institute
4676 Admiralty Way
Marina del Rey CA 90291

## 1. A DEVELOPMENT METHODOLOGY

At ISI we are researching a methodology for assisting software development. It is our firm belief that to make significant progress in this direction we must formalize, record and manipulate the development process itself. The methodology we advocate is one of constructing a formal specification expressing desired (functional) behaviour, and then transforming this into an implementation to achieve efficiency whilst preserving functionality.

What differentiates our research from that of others pursuing this transformational approach is the nature of our specification language. This has been designed to minimize the distance between informal conceptualization of tasks and formal specifications of the same. A consequence of this is that the richness of our specification language makes automatic compilation into tolerably efficient programs beyond our present capability (and to restrict ourselves to the use of only those specification constructs that we can presently compile would, we feel, be a grave mistake for our or any other specification language). Hence the transformation from specification to implementation must rely upon human guidance (although it can and should benefit from machine assistance to record and perform the detailed steps).

Within this methodology reuse may occur at two levels:

- Reuse at the specification level - our specification language comprises a small set of powerful constructs which are used in stylistically recurring ways in specifying a broad range of tasks. Reuse also occurs when a specification is modified, either to take into account desired changes inspired by feedback from the implementation, or to adapt that existing specification to a similar task.

- Reuse at the development level - when instances of the high level specification constructs have to be transformed into efficient (implementation) constructs: stylistically similar uses of such constructs give rise to the same range of issues in selecting an appropriate implementation and in application of transformations to map them into their implementations. Reuse also occurs when a modified specification has to be redeveloped into an implementation, or altered requirements for efficiency permute the various efficiency tradeoffs, and so would inspire different choices during the development of an implementation of the same specification - in both cases reuse of some of the original development may take place

We stress the importance of performing modification and maintenance on the specification, and then redeveloping the implementation from that, as opposed to tinkering with the implementation directly.

Reuse arising from modification of a specification and its subsequent reimplementation comprise those aspects of reuse with which we have had the least experience, and form an area of research we intend to pursue in the near future - see the companion paper by Balzer for details in this regard. The other forms of reuse - that is, recurring themes in using high level specification constructs and in transforming them into implementations - comprise the focus of this paper. We report on the achievements we have made in these issues, i.e., what are the features of our specification language that support reuse at the conceptual level, and what are the techniques we have accumulated to convert such specifications into implementations.

## 2. SPECIFICATION LANGUAGE

Our specification language, Gist, supports the description of the *behaviour* required of a process. The motivating source of Gist's capabilities is the power of natural language descriptions: Gist attempts to provide constructs to capture this power in a formal language. Briefly, these capabilities are as follows:

- *a relational and associative model of data*: which captures the logical structure without imposing an access regime,

- *information derivation*: which allows for global declarations describing relationships among data,

- *historical reference*: the ability to refer to past process states,

- *constraints*: restrictions on acceptable system behaviour in the form of global declarations,

- *demons*: asynchronous processes responding to defined stimuli, and

- *closed system specification*: the ability to implicitly describe the behaviour of a portion of some larger system by describing the behaviour required of the whole system and the interface between that portion and the remainder.

The primary design goal behind Gist is to minimize the gap between informal conceptualization and formal specification. This enables us to formally capture as much as possible of the

development process, particularly the early stages, which would otherwise have to be performed entirely within the heads of designers, going unrecorded and unassisted by machine tools. Gist specifications are free of implementation concerns such as efficiency, data representation and algorithms, instead their emphasis is on describing required behaviour. Gist's capabilities support this *descriptive* (as opposed to prescriptive) style. Gist specifications tend to *localize* the expression of features of the required behaviour (as opposed to implementations, which achieve optimality by spreading information and control throughout the program in order to share data and activity). These aspects combine to make Gist supportive of both initial specification, and also of later modifications to the specification. When modification takes the form of adding further detail, the existing specification tends to be "robust" in the sense that it should require little or no change when additions do force some change or when the modification is itself a change (to the existing specification), the descriptive and localized aspects make it easy to identify the impact of the modifications on the specification, and easy to perform the appropriate adjustments. In contrast, a less expressive specification language would force the mental conversion of changes at the conceptual level into changes at a lower, more implementation-oriented, level, with the dual disadvantages of being harder (more mental effort, less opportunity for machine support - hence more error prone) and failing to record some of the development process (less documentation - hence less comprehensible, and less of a basis for supporting future change).

## 3. DEVELOPMENT OF IMPLEMENTATION

Development of an implementation from a Gist specification necessitates the elimination of uses of Gist's high-level constructs, since their success in the realm of specification is at the expense of freedom from concerns such as efficiency, data representations and algorithms. Program transformation is the means by which we perform such development.

We prefer to seek transformations that deal at once with whole instances of these constructs, as opposed to unfolding instances into lower level primitives. In adopting this approach we abandon the hope for a small set of simple transformations, but retain the advantage of dealing with the constructs at as high a level as possible.

For each type of construct, our research has been aimed at accumulating:

- *implementation options*: commonly available options for converting an instance of that construct into a more efficient expression of the same behaviour, typically in terms of lower level constructs;
- *selection criteria*: for selecting among several implementation options applicable to the same instance, and
- *mappings*: to achieve the implementation options via sequences of equivalence preserving transformations. E.g., a (historical) reference to the time-ordered sequence of objects to satisfy some predicate may be mapped into a data structure that explicitly stores that sequence, code to append objects to that sequence as they begin to satisfy the predicate, and code to replace the historical reference with a simple retrieval from the data structure.

Our experience on small examples has shown that these are of recurring use in the development process. Samples of our findings serve to illustrate both the features of Gist and the transformational techniques that we have accumulated to deal with them.

## 4. AN EXAMPLE DOMAIN

In the next section we will use examples drawn from the domain of a single problem to illustrate our approach. The problem we choose is a routing system for distributing packages into destination bins. This problem was constructed by representatives of the process control industry to be typical of their real-world applications. Hommel's study of various programming methodologies used this problem as the comparative example [Hommel 80].

The figure below illustrates the routing network. At the top, a source station feeds packages one at a time into the network, which is a binary tree consisting of switches connected by pipes. The terminal nodes of the binary tree are the destination bins.

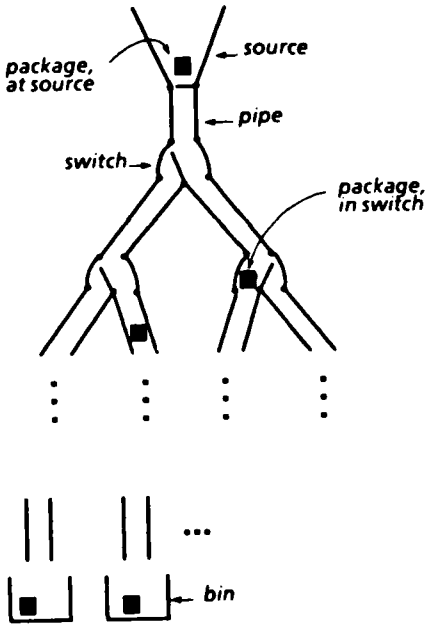



**Figure 4-1**: The package router

When a package arrives at the source station, its intended destination (one of the bins) is determined. The package is then released into the pipe leading from the source station. For a package to reach its designated destination bin, the switches in the network must be set to direct the package through the network and into the correct bin.

Packages move through the network by gravity (working against friction), and so steady movement of packages cannot be guaranteed; they may "bunch up" within the network and thus make it impossible to set a switch properly between the passage of

two such bunched packages (a switch cannot be set when there is a package or packages in the switch for fear of damaging such packages). If a new package's destination differs from that of the immediately preceding package, its release from the source station is delayed a (precalculated) fixed length of time (to reduce the chance of bunching). In spite of such precautions, packages may still bunch up and become misrouted, ending up in the wrong bin: the package router is to signal such an event.

Only a limited amount of information is available to the package router to effect its desired behaviour. At the time of arrival at the source station but not thereafter, the destination of a package may be determined. The only means of determining the locations of packages within the network is a group of sensors (placed on the entries and exits of switches and on the entries of bins); these sensors detect the passage of packages but are unable to determine their identity. (The sensors are able to recognize the passage of individual packages, regardless of bunching).

# 5. Gist's constructs

In this section Gist's major constructs are considered in turn, and for each we:

- informally describe the semantics of the construct, with illustrations from the package router domain,
- describe the freedoms the construct provides for specification,
- discuss how the construct supports the expression of changed versions of a specification when incorporating modifications: hypothetical modifications to the package router serve as illustrations, and
- briefly describe some alternative mappings for the construct, together with criteria for choosing among these alternatives.

## 5.1. Relational and associative model of information

We begin the discussion of Gist's major features by focusing on its underlying data model. Information in Gist is modeled simply by typed objects and relations among them.

> *The package router domain involves objects of type package, objects of type switch, etc. Type hierarchies are possible: for example, a switch or a bin might more generally be considered a location.*

Relations among these objects model information about this domain.

> *The structure of the network is modeled by relations between locations - i.e., the connection between source and first pipe, the connection between that first pipe and the switch to which it leads, etc., will all be modeled by relations between those objects. The position of a package is modeled by a relation between that package and its location; the setting of a switch (i.e., the outlet pipe into which the switch is currently directing packages) is modeled by a relation between switch and pipe.*

The collection of objects and relations at any time during the interpretation of a specification comprises what we call a "state". Change in the domain is modeled by the creation and destruction of objects and by the insertion and deletion of relations. Each change is a transition from the current state into a new state. Multiple changes may occur simultaneously in a single transition from one state to the next.

> *The arrival of a package at the source is modeled by inserting the position relation to hold between that package and the source location.*
>
> *Altering the setting of a switch is modeled by changing the switch-setting relation, i.e., deleting the relationship between switch and old setting, and inserting the relationship between switch and new setting.*
>
> *Simultaneous movement of two packages is modeled by changing their position relations in the same transition.*

### 5.1.1. Specification freedom

The relational model of information permits the specifier to use a descriptive reference to an object to refer to that object.

> *The bin that is the destination of this package.*
>
> *The pipe into which this switch is set to direct packages.*

The relational data model is a very general data representation. The specifier need not be concerned about data access paths, for instance, because any description of an object may be used as a reference to that object. Relations may be used in descriptions of any of the objects that participate in the relationships. In data base terminology, this means that the relationships are fully associative (or, equivalently, that the data base is fully inverted).

> *The position relation (between package and location) may be used in describing a location: "The location that is the position of this package" and in describing a package: "The package(s) whose position is this location".*

Concern about the statistical distribution of these operations is unnecessary. The implementation process selects particular physical representations for information that are appropriate for anticipated patterns of data storage and access.

### 5.1.2. Specification reuse

The generality of the relational model, and the freedom from representation concerns that it provides, facilitate the expression of changed specifications.

> *A refinement to the router to check that packages have sufficient postage stamps to pay for delivery to their respective destinations involves extra information -- stamp values on packages, and required stamp values for destinations. Modeling this extra information is achieved by defining a new type, stamp values, and additional relationships, between packages and stamp values, and between destinations and required stamp values.*

### 5.1.3. Mappings

The most general solution to implementing information storage is to support an associative relational data-base and leave the specification's insertions and retrievals of information unchanged. In most cases, however, a specification does not indiscriminately insert or retrieve data; rather, it displays predictable data access patterns. These can be mapped into appropriate data structures

(arrays, hash tables, etc..) to conserve space and time.

> *If the relation that models the destination of a package (i.e., a relation between the package and the bin that is its destination) is accessed in one direction only, by asking for the bin that is the destination of a package, then the destination information could be stored as a field of a record structure of information associated with each package*

Concerns for efficiency of time and space dictate the selection of data structures. Probabilistic expectations of frequency of use are not explicitly described in Gist specifications. Clearly, for implementation purposes such information will be of importance in selection.

It should be noted that many of the issues relating to the relational data model are similar to those investigated by the SETL group [Dewar et al 79], [Schonberg, Schwartz & Sharir 81] and by Rovner [Rovner 78], Low [Low 76], and Barstow [Barstow 79].

### 5.2. Information derivation

Often it is convenient to make use of a relationship that is derived from other relationships. Within Gist the derivation of such a relationship may be declared once and for all, and serves to denote all the maintenance necessary to preserve the invariant between the derived relation and the relations upon which it depends.

> *A switch may be said to be empty if there is no package whose location is the switch (hence 'empty' is a unary relation).*
>
> *A location may be said to be 'below' a second location if it is immediately below that second location, or is immediately below some third location that is in turn 'below' the second location (i.e., transitive closure of immediately below).*

#### 5.2.1. Specification freedom

The specificational power of this construct comes from being able to state a derivation in a single place (i.e., this construct exhibits the quality of "locality") and then make use of the derived information throughout the specification. As with explicitly inserted relations, data access is fully associative.

> *The derived 'below' relation may be accessed in either direction, i.e., given a location, the relation may be accessed to find either the locations which are 'below' that location, or the locations which that location is 'below'.*

#### 5.2.2. Specification reuse

Derived relations provide robustness in the face of specification change, both because of their localised nature, and because they are defined in terms of the information upon which they depend (i.e., have the "descriptive" quality).

> *Should the structure of the package router network be extended by addition of more pipes, switches and bins, the definition of the derived relations 'empty' and 'below' will continue to be valid.*

On the occasions when the definitions of derived relations must be modified, the localised nature of their definitions eases the task of correctly making such modifications.

#### 5.2.3. Mappings

Since no corresponding construct is likely to be available in any implementation language we might choose[1] we must map the derivation into explicit data structures and mechanisms to support all the uses of that information scattered throughout the program. We have a wide range of choices as to how we might do this mapping.

At one extreme, we might simply unfold the derivation at all the places where a reference to the relation is made. Having done this, we may completely discard the relation and its derivation.

> *Wherever the specification makes reference to the 'empty' relation on a switch, unfold the definition of 'empty', to leave in its place an explicit search through all the packages to determine whether any of them are located at the switch.*

This approach is analogous to backward inference, where computation is performed on demand and at the site of the need.

At the other extreme we might retain the relation, but distribute throughout the program the code necessary to explicitly maintain the invariant between the derived information and the information upon which it depends.

> *To maintain the derived relation of switch 'empty', introduce explicit storage (in the form of a non-derived relation) to represent this information, and introduce the appropriate maintenance code everywhere in the specification that the locations of packages might change (more precisely, at the places where a package may become located at, or cease being located at, switches).*

*This approach is analogous to forward inference, where* computation is performed whenever a modification to a relevant predicate occurs and at the site of the change. There are two separate capabilities required by this mapping:

1. determining all those locations in the specification at which the value of a derived relation could possibly be changed, and
2. inserting code to do the recalculation at those locations.

The latter capability can be achieved by either recomputing the defined relation from scratch, or incrementally changing its present value.

> *To maintain the sequence of packages in a pipe, when a package enters the pipe, concatenate that package onto the end of the maintained sequence; when a package exits, remove the package from the front of the maintained sequence.*

This is an example of a general technique we call "incremental maintenance", and is derived from the work of other researchers in set-theoretic settings, particularly [Paige & Schwartz 77], who

---

[1] Many of the Artificial Intelligence programming languages do provide facilities for implementing derived relations in terms of inference processes. For example, a implementation of derived relations might be provided in CONNIVER [McDermott & Sussman 74] in terms of IF-ADDED or IF-NEEDED methods. However, AI programming languages in which these facilities are present typically do not provide for the efficient execution one would desire for an optimized implementation, nor do these facilities provide precisely the semantics desired without the inclusion of satisfactory "truth maintenance" capabilities, [Doyle 79], [London 78].

call the technique "formal differentiation", and [Earley 75], who calls it "iterator inversion".

Unfolding a derived relation results in rederivation at points of use: maintaining it results in rederivation (incrementally or otherwise) at points of change. It is permissible to do the computation for maintaining the relation at other points, but it must have its correct value by the time is used.

The choices among the implementation alternatives suggest alternatives between storage and computation in the resulting program. Completely unfolding the derivation is tending towards complex recalculation with a minimum of stored data. Maintenance simplifies retrievals at the expense of the maintenance operations and the extra storage to hold the maintained information.

### 5.3. Historical reference

Historical reference in Gist specifications provides the ability to extract information from any preceding state in the computation history.

> *Has this package ever been at that switch?,*
>
> *What was the most recent package to have been in this switch?,*
>
> *Was the bin empty when the package entered the network?*

Note that the past can only be queried, not changed.

#### 5.3.1. Specification freedom

Historical reference allows the specifier to easily and unambiguously describe *what* information is needed from earlier states without concern for the details of *how* it might be made available (i.e., like derived information, this construct has the "descriptive" quality). Reference to the past has been studied in the database world, where the freedom has been called "memory independence", and temporal logic has been applied to formally investigate the matter (see, e.g., [Sernadas 80]). Both constructs may be mixed, using derived relations in expressing a historical reference, and using historical reference in defining a derived relation.

> *Historical reference: "Was this switch ever 'empty'?".*
>
> *Derived relation definition. "The sequence of packages to have been located at the source, in their order of arrival there".*

This exemplifies one of Gist's strengths, the "orthogonality" of the constructs, i.e., they may be successfully used in combination.

#### 5.3.2. Specification reuse

Historical reference, like derived information, provides robustness as a consequence of its descriptive nature, in this case, robustness in the face of modifications that result in changed histories.

> *Should the topology of the network be modified, say to feed the output of several pipes into the same bin, then the descriptive historical reference "The sequence of packages to have reached the bin in their order of arrival" will continue to be valid*

#### 5.3.3. Mappings

Two generally applicable methods exist for mapping historical reference into a reasonable implementation. These are:

1. save the information desired in the earlier state, then modify the historical reference to extract it from the saved information, or
2. modify the historical reference to rederive the desired information from the current state.

To use the first method, it is necessary to introduce and maintain auxiliary data structures to store information that might be referenced in a later state, and modify the historical references to extract the desired information from those introduced structures when needed. The desire for economy of storage in an implementation encourages the implementor to determine just what information need be preserved, seek a compact representation facilitating both storage and retrieval, and discard the information once it is no longer useful.

> *To be prepared to answer the query "what was the destination of the last package to have passed through this switch?" we could choose to remember the time-ordered sequence of packages to have been in the switch, or more efficiently, only the destination of the immediately preceding package. This latter case would require storage space for the identity of only a single destination bin; upon arrival of a new package, the identity of its destination would be remembered in that space, overwriting the old information.*

The alternative method for implementing historical reference is to rederive the desired information in terms of information *available in the current state* (without having to retain extra information from past states).

> *The identity of the previous package to have been at this switch might be derived by determining which package in the network is closest to and downhill from the switch.*

We suspect that rederivation is rarely an available option; the information desired is often not derivable from current available information. When both options are possible, they present the classic store/recompute tradeoffs. An implementor must compare the cost of the derivation with the cost of storage and maintenance of redundant information to permit simple access.

#### 5.3.4. Idiomatic uses of historical reference

Certain patterns of historical reference recur frequently in Gist specifications. For example, evaluating *<predicate> asof <event>*, or *<expression> asof <event>* (of which *What was the setting of the switch at the time the package entered the network?* is an example). For an idiom like this we can construct special purpose mappings,[2] reducing the effort that would be required during implementation development if a general purpose mapping technique was applied. A general-purpose mapping technique would require application of further simplifications to tailor the result for the special case.

---

[2] This idiom is mapped into an explicit relation between the objects that parameterize the event and the <predicate> / <expression>, together with code to maintain this relation, namely to insert the relation whenever the event occurs and the <predicate> holds / there exists an object denoted by the <expression>

Other idioms that we deal with include:

- the latest object to satisfy a given predicate,
- the sequence of objects ordered by their time of creation or the time at which they satisfied a given predicate,
- did event$_1$ take place before event$_2$?

### 5.4. Nondeterminism and constraints

Nondeterminism within Gist occurs in two ways: When use is made of a descriptive reference that denotes more than one object.

> *Set the switch to one of the switch outlets.*

or when some specifically nondeterministic control structure is used

> *Choose between "Set switch" and "Release package".*

In terms of the behaviour that a Gist specification denotes, nondeterminism gives rise to a set of behaviours: an implementor is free to select any (non-empty!) subset[3] of those behaviours as the ones his implementation will satisfy.

> *The activity of setting switches is described nondeterministically by stating that at random times random switches set themselves to a random one of their outlet pipes.*

Constraints within Gist provide a means of stating integrity conditions that must always remain satisfied.

> *Packages in some location cannot overtake one another (i.e., a package that entered some location later than another package cannot leave that location before that other package).*
>
> *A switch must be empty in order to change its setting.*
>
> *A package must never reach a wrong bin (i.e., some bin other than its destination)*[4]

Within Gist, constraints are more than merely redundant checks that the specification always generates valid behaviours; constraints serve to *rule out* those behaviours that would be invalid.

> *The nondeterminism of switch setting, in conjunction with the constraint on packages reaching correct bins, denotes only behaviours that route the packages to the proper destination bins.*

#### 5.4.1. Specification freedom

The constructs described in previous sections provided freedoms related to information: nondeterminism and constraints, and as we shall see, demons too, provide freedoms related to control.

Where there are several equally acceptable alternatives in the resolution of a data reference or a control structure choice, nondeterminism makes it easy to express them all. Where there are integrity conditions that must be satisfied, constraints provide a concise (i.e., localised) means of stating them. Such integrity conditions may serve as descriptions of the environment in which the portion to be implemented is to operate, and so provide information about the environment upon which the implementor may rely (e.g., the non-overtaking of packages within locations is a property of the physical routing mechanism). Other integrity conditions serve as requirements on the behaviour of the system, implying that the implementor must implement his portion in such a manner that it will operate with the environment to satisfy those conditions (e.g., that the switch be empty in order to change its setting).

The conjunction of nondeterminism and constraints provides an extremely powerful specification technique: a specification denotes those and only those behaviours that do not violate constraints. In contrast, an implementation is characterized by the cunning encoding of its components to interact in ways guaranteed to result in only valid behaviours.

#### 5.4.2. Specification reuse

Constraints provide robustness in the face of specification change because by their very nature they guarantee that all the behaviours (old or new) denoted by a changed specification must abide by all the constraints that remain in, or have been added to, the specification.

> *The constraint that a switch be empty in order to change its setting assures us that no matter how the topology of the network might be modified, we may remain assured that no new behaviour will result in which a switch setting changes while some package is present in that switch.*

Constraints themselves are readily modified to reflect changing criteria.

> *To further restrict when a switch setting may be changed, say to only those occasions when that switch and the pipe that leads into it are both empty, we simply modify the constraint accordingly - a single modification at only one place in the specification.*

#### 5.4.3. Mapping away constraints and nondeterminism

A general mapping technique to eliminate constraints is to make each nondeterministic activity into a choice point, and to unfold global constraints so as to provide tests at all points in the program where the constraint might possibly be violated. When a violation is detected, a "failure" results: this causes backtracking to the most recent choice point with an alternative choice.

> *To place 8 queens on a chess board under the constraint that no queen may attack any other queen, simultaneously place all the queens on the board ($64^8$ nondeterministic choices!), and after doing so, check to see whether the no-capture constraint is violated - if so, try the next choice of placements. [Note that this is a most inefficient mapping.]*

This mapping is similar to the maintenance mapping for derived relations: two separate processes are required to implement the mapping:

---

[3] I.e., there may be behaviours denoted by the specification not displayed by the implementation. Conversely however, any behaviour displayed by the implementation must be one of the behaviours denoted by the specification.

[4] Although this would be a desirable constraint to impose on the package router, it would render an implementation impossible because of the conditions within which the router mechanism has to operate, most notably the vagaries of the movement of packages, and the limits on switch setting. It makes a nice example, though.

1. determining all locations in the specification at which the constraint might be violated (similar to determining all locations where the value of a derived relation could change), and

2. inserting, at those points, code to do the checks and backtracking (whereas in the case of derived relations, the code inserted would have the purpose of updating the stored value of the relation).

The second capability mentioned above as required for mapping constraints and nondeterminism implements the backtracking control mechanism. Our research suggests that it is possible to intermix Gist's nondeterminism and constraints with explicit backtracking, permitting the incremental mapping of individual nondeterministic choice points, and of the constraints that impinge upon them (as opposed to having to simultaneously map away *all* the nondeterminism and constraints at once). The task of building the backtracking mechanism itself is trivial if our intended target language supports backtracking (as does, for example, PROLOG [Clocksin & Mellish 81]).

Backtracking, however, presupposes the ability to undo actions that have been executed since the last choice point. Since this is often not possible, strict backtracking is not always an option for mapping to an implementation of nondeterminism.

> *In controlling the switches in the routing network, we might be constrained to ensure that the packages do not reach wrong destinations. Backtracking presupposes that we have the ability to return the packages to the switching points after an error is detected and then send them in different directions. Obviously, in the case of a package router whose package movement mechanism is not under our control, this is not an available option.*

An alternative technique to mapping nondeterminism into backtracking is a "predictive" solution. Here, the constraints are unfolded, but into "point" constraints rather than into calls on a backtracking failure mechanism. These point constraints are then pushed back to be incorporated into the choice points, becoming filters that propose only those choices that guarantee no constraints will be violated. "Pushing" a point constraint backwards over a statement is a matter of reformulating the constraint into the weakest precondition to that statement that guarantees execution of the statement will not violate the constraint. When the constraint is "pushed" all the way back to a choice point, it is incorporated as a filter on the choices.

> *When a switch becomes empty, choose to set it in the direction that leads to the destination of the next package approaching the switch.*

Compromise between these two extremes is possible: we may employ a backtracking algorithm, but push some (though not all) of the unfolded constraint(s) into the choice generators.

> *In the 8-queens problem, split the nondeterminism into several successive choices (place the first queen, place the second queen and check for capture, etc.), and incorporate some of the no-capture constraint into the placement (by not attempting to place a subsequent queen on a row already occupied by a queen). See [Balzer 81] for a detailed development illustrating this.*

The choice between a backtracking implementation and a predictive implementation is determined very much by the nature of the domain of the specification. The capabilities of, and the control we have of, the effectors[5] (if there are any in the system being specified), the amount of information available for making decisions, and the desired amount of precomputation all affect the choice of algorithm. Typically, the interesting issue is to develop the specification toward an implementation that embodies an algorithm to perform the search efficiently, and not to assume that the result has been pre-calculated and make use of it.

### 5.5. Demons

Demons provide Gist's mechanism for data-directed invocation of processes. A demon has two components a *trigger* and a *response*. Whenever a state change induces a change in the value of the trigger predicate from false to true, the demon's response is invoked.

> *Whenever a package reaches a wrong bin (some bin other that its intended destination), send a signal*

#### 5.5.1. Specification freedom

Demons are a convenient construct for situations in which we wish to specify the execution as an asynchronous activity that arises from a particular change of state in the modeled environment. This eliminates the need to identify individual portions of the specification where actions might cause such a change and the need to insert into such places the additional code necessary to invoke the response accordingly. The specificational power of the demon construct is enhanced by the power of Gist's other features: for example, the triggering predicate may make use of derived information.

#### 5.5.2. Specification reuse

The descriptive nature of demons -- i.e., the description of the condition upon which some activity is to be started -- provides the robustness. Should modifications to the specification change the behaviours that may occur, the demons will continue to be triggered when and only when their triggering conditions are met.

> *Should the topology of the router network be modified so that several pipes lead to the same output bin (i.e., the network is no longer a tree), then the demon that signals arrivals of misrouted packages will continue to do its signaling regardless of which pipe they happen to emerge from.*

#### 5.5.3. Mapping away demons

Mapping away a demon involves identifying all places in the program where a state change might cause a change of the value of the demon's trigger from false to true, and then inserting code at those places to make the determination and perform the demon's response when necessary.

> *To map away a demon which sends a signal every time a package reaches a wrong bin, introduce code into the places where package movement occurs to check to see whether that package has moved into a bin other than its destination; in such a case, perform the signalling.*

---

[5] E.g., in a routing network, the switches and conveyor

This is another mapping that makes use of the capability to identify locations in the specification where the value of a predicate (in this case the demon's trigger) might be affected (in particular, change from false to true). Here the action to be taken upon detecting such a change is to invoke the demon's response.

Demons are a potent source of nondeterminism in the behaviours denoted by a specification. This is because our semantics for demon "response" are that a new "line of control" to perform that response is begun, and that line of control runs in parallel with the already active line(s) of control, permitting any arbitrary interleaving that does not violate constraints.

> *In the package router, specify the behaviour of a switch via a demon that has a random trigger[6], and whose response is to set the switch to any of its outlets (further nondeterminism). Together with a constraint to prohibit items from being routed to incorrect destinations, this would suffice to denote behaviours in which switches are set at the appropriate times and in the appropriate directions to effect correct routing, while denoting complete freedom of switch behaviour when their settings are not crucial to the routing of any packages.*

Our experience with both specifying and mapping this form of nondeterminism is somewhat limited. We anticipate that in many cases it will be preferable to map this form of nondeterministic control structure while expressed concisely as demons, rather than first to unfold those demons throughout the specification and then to have to manage the resulting disparate instances of nondeterminism. Our hope is that control nondeterminism provided by demons, constrained by constraints which prune out the undesired effects of arbitrary interleaving, will occur in commonly occurring styles of usage, for which we will be able to build idiomatic mappings.

### 5.6. Closed system style

For specification purposes it is convenient to describe the behaviour required of an entire system, and to build that description in terms of information throughout the system. How that entire system is to be decomposed into components, the restrictions on the control that components may exert over one another, and the access that components may have to each others information, may described separately from the overall system behaviour description.

> *In the package router, we describe the behaviour required of the overall system, namely the routing of packages. This description is expressed in terms of package locations and destinations. Separately, we describe how the system comprises several components:*
>
> - *physical routing mechanism (the binary tree of source, pipes, switches and bins),*
> - *an uncontrollable and unpredictable mechanism to move packages (gravity interacting with friction),*
> - *an unpredictable input of packages at the source to be routed, and*
> - *the switch controller which may change switch settings.*
>
> *It is our development task to implement the switch controller so as to interact with the other components in such a way as to cause the desired routing of packages. Furthermore, the switch controller has only limited control of, and access to, the other components. A switch setting may only be changed when it is empty of packages. The only occasion upon which the controller mechanism may read the destination of a package is when that package is at the source. Once released into the network, the only information available to the controller is the passage of individual packages past sensors, and even then only the face that some package has passed is available, not even the identity of the package.*

#### 5.6.1. Specification freedom

Closed system description provides the freedom to describe the behaviour required of the whole system, and to give this description in terms of system-wide information. The decomposition of the system into components is specified separately. The behaviours of these components are thus implicitly defined to be those which in conjunction will achieve the required system wide behaviour while complying with the limitations on control and information passing between components. Maximum freedom is left to the developer to choose any of these implicitly defined behaviours for the components to be implemented.

#### 5.6.2. Specification reuse

Since system wide behaviour is specified directly, modifications to that behaviour are easy to incorporate. Modifications to the environment (within which the implemented portion resides) may also be readily expressed, and the system-wide specification of required behaviour remains unchanged.

> *If the physical router mechanism is adjusted to move packages by means of fixed-speed conveyor belts so as to prevent packages from bunching up, this extra property may simply be added to the specification of the environment. The implementor will then be able to take advantage of this extra information in his rederivation of an implementation (and in such a case could guarantee 100% correct routing).*

Having the decomposition stated separately from system behaviour -- indeed, having it explicitly stated at all -- supports modifications to the decomposition.

> *If the sensors throughout the package router network are enhanced to report not only the passage of a package, but also the destination of the passing package, then this enhancement may be incorporated into the specification of the decomposition, permitting the implementor of the router mechanism to make use of the extra information (which should result in a reduction in the amount of storage space for data required by the implementation).*

#### 5.6.3. Mapping away reliance on system-wide control and information

The general problem of mapping away such reliance is quite difficult. Mostow calls this aspect of development "operationalization", and has investigated heuristic means for dealing with it, [Mostow 81].

---

[6] Indicating that in any state each such demon has the nondeterministic choice of triggering or not triggering.

For some simple cases of reliance on system-wide information, techniques similar to those used for mapping away historical references might prove appropriate - introduce and maintain auxiliary data structures to hold information when it is made available in order to be able to supply that information when it is needed, or look for ways to derive the required information from other information that is available.

## 6. SUMMARY AND FUTURE WORK

Gist's combination of constructs makes for a good specification language, and, we anticipate, will support reuse at the specification level. We attribute Gist's success to its purposeful design -- modeled on the power of natural language descriptions, brought together into a coherent formal framework.

The development methodology we advocate is one of transformation of specifications to obtain implementations. The success or failure of this methodology for supporting reuse rests upon our ability to reperform the transformational development upon a modified specification. This is the crucial outstanding research issue.

The mappings for Gist constructs will comprise the primitive steps from which Gist developments will be composed, and the choice criteria between mappings will form the basis for selection of appropriate mappings. It has been recognised that transformational developments must be objects in their own right, so that they may be applied to specifications to produce implementations, and appropriately modified to be applied to modified specifications [Darlington & Feather 80]. The language for recording such developments must be rich enough to capture the implementor's goal structure -- motivations and design decisions -- [Sintzoff 80] and [Wile 82]. The system that applies such developments must deal with this goal structure, methods for achieving goals, and selection criteria for choosing among competing methods, and, for the foreseeable future, will not be fully automatic, so must rely upon interaction with a skilled implementor. See [Fickas 82] for a first cut at such a system.

### Acknowledgements

This research was supported by Defense Advanced Research Projects Agency contract MDA 903 81 C 0335. Views and conclusions contained in this document are those of the author and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, or any other person or agency connected with them. I would like to thank the other members (past and present) of the ISI Transformational Implementation group: Bob Balzer, Don Cohen, Steve Fickas, Neil Goldman, Phil London, Jack Mostow, Bill Swartout and Dave Wile.

## References

[Balzer 81] Balzer, R. "Transformational Implementation. An Example." *IEEE Transactions on Software Engineering* SE-7, (1), 1981, 3-14.

[Barstow 79] Barstow, D.R., *"Knowledge-Based Program Construction"*, Elsevier North-Holland, 1979.

[Clocksin & Mellish 81] Clocksin, W.F. & Mellish, C.S., *Programming in Prolog*, Springer Verlag, Berlin, 1981.

[Darlington & Feather 80] Darlington, J. & Feather, M.S., *"A transformational approach to program modification"*, Department of Computing and Control, Imperial College, London, Technical Report 80/3, 1980.

[Dewar et al 79] Dewar, R.B.K., Grand, A., Liu, S. & Schwartz, J.T., "Programming by refinement, as exemplified by the SETL representation sublanguage," *ACM Transactions on Programming Languages and Systems* 1, (1), 1979, 27-49.

[Doyle 79] Doyle, J., "A truth maintenance system," *Artificial Intelligence* 12, (3), 1979, 231-272.

[Earley 75] Earley, J., "High level iterators and a method for automatically designing data structure representation," *Computer Languages* 1, (4), 1975, 321-342.

[Fickas 82] Fickas, S.F., *"Automating the transformational development of software"*, Ph.D. thesis, University of California, Irvine, 1982.

[Hommel 80] Hommel, G., *Vergleich verschiedener Spezifikationsverfahren am Beispiel einer Paketverteilanlage*, Kernforschungszentrum Karlsruhe, Technical Report, August 1980.

[London 78] London, P., "A dependency-based modelling mechanism for problem solving," in *AFIPS Conference Proceedings, Vol. 47*, pp. 263-274, 1978.

[Low 76] Low, J.R., *Interdisciplinary Systems Research*, Volume 16: *"Automatic Coding: Choice of Data Structures"*, Birkhauser Verlag, Basel & Stuttgart, 1976.

[McDermott & Sussman 74] McDermott, D. & Sussman, G. J., *The CONNIVER reference manual*, MIT, Technical Report Memo 259a, 1974.

[Mostow 81] Mostow, D.J., *"Mechanical transformation of task heuristics into operational procedures"*, Ph.D. thesis, Computer Science Department, Carnegie-Mellon University, 1981.

[Paige & Schwartz 77] Paige, R. & Schwartz, J., "Expression continuity and the formal differentiation of algorithms," in *Proceedings, 4th ACM POPL Symposium, Los Angeles*, pp. 58-71, 1977.

[Rovner 78] Rovner, P., "Automatic representation selection for associative data structures," in *Proceedings, AFIPS National Computer Conference, Anaheim, California*, pp. 691-701, AFIPS Press, New Jersey, June 1978.

[Schonberg, Schwartz & Sharir 81] Schonberg, E., Schwartz, J.T. & Sharir, M., "An automatic technique for selection of data representations in SETL programs," *ACM Transactions on Programming Languages and Systems* 3, (2), April 1981, 126-143.

[Sernadas 80] Sernadas, A., "Temporal aspects of logical procedure definition," *Information Systems* 5, (3), 1980, 167-187.

[Sintzoff 80] Sintzoff, M., "Suggestions for composing and specifying program design decisions," in *4th International Symposium on Programming*, Paris, April 1980.

[Wile 82] Wile, D. S., *Program developments: formal explanations of implementations*, ISI, 4676 Admiralty Way, Marina del Rey, CA 90291, Technical Report RR-82-99, 1982. To appear in CACM