Mappings for Rapid Prototyping

Martin S. Feather

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

27 September 1982

Abstract

The transformational methodology for software development is adapted to perform rapid conversion of specifications into prototypes. This makes feasible testing of specifications to observe their behaviours and assuring that specifications can indeed be implemented.

The approach is centered on gathering techniques to map each type of specification language construct into a reasonably efficient implementation. Instances of these constructs in an actual specification may then serve as the focal points for the conversion process.

This research was supported by Defense Advanced Research Projects Agency contract DAHC15 72 C 0308. 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.

1. Introduction

At ISI we are researching a methodology for software development in which a formal specification expressing desired (functional) behaviour is first constructed, and then transformed into an implementation to achieve efficiency whilst preserving functionality. This methodology is readily adapted to rapid prototyping; the first step, construction of a formal specification, remains unchanged. Details of our approach to specification may be found in the companion paper, [Balzer et al 82]. Having obtained a formal specification, it remains to investigate its implied behaviour, to ensure that it indeed captures our intent. The two ways that we see this might be done are to

- symbolically execute the specification (we are in the process of constructing a symbolic executer to do just this - see the companion paper, [Cohen et al 82] for details), or
- quickly convert the specification into an implementation that may be fed test cases on which to observe its behaviour.

It is efforts toward the latter that are the focus of this paper.

Since the constructs of our specification language, Gist, are tailored for ease of expression rather than ease of implementation, we find that a naive interpretation of them would be grossly inefficient, incapable of handling even small test cases. Hence unless we can build a sufficiently capable (sophisticated) interpreter, we must rely upon developing equivalent implementations as a prerequisite to testing. Even after construction of such an interpreter, this alternative remains viable, as it would still speed the testing process.

To meet the needs of rapid prototyping, we require that the development of a testable implementation from a specification be both correct and rapid. By the former we mean that the development process assuredly preserve the functional behaviour implied by the specification. To achieve this we use equivalence preserving transformations as our means of manipulating specifications into implementations, and rely upon machine support to perform the transformation applications. Hence the need for rapid development translates into the need for rapidly finding a sequence of transformations that will effect conversion of a specification into an implementation with tolerable efficiency for testing purposes (which will be far easier than achieving an optimal implementation).

We might suppose that if the semantics of each of Gist's constructs were defined by transformation in terms of a kernel language (in the manner that the CIP group in Munich define their wide spectrum language, [Bauer et al 78]), then it would suffice to apply these transformations to achieve a testable

prototype. Not surprisingly, the general purpose semantic definitions are most unlikely to be suited to producing even tolerably efficient prototypes. Converting into a kernel of the language and then applying efficiency transformations would also be a mistake. As has been recognised by past researchers, it is far better to perform optimisations on high-level constructs at the level of those constructs rather than convert them into a lower level and then hope to achieve the same effect via lower level transformations.

Our proposed solution to finding the sequence of transformations is to use instances of specific Gist constructs, which provide freedoms from implementation concerns, as the focal points on which to base our development activities. For each such specification construct we formulate:

- *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.

When presented with a specification we may focus upon instances of specification constructs, and for each identify applicable mappings, use the selection criteria to pick one, and apply the transformations to effect it.

An advantage of this approach is that it is amenable to incrementally adding automated support as we gain increased understanding of mappings and their associated techniques. Prior to gaining such understanding, we may use machine support to do the low level tasks of applying the transformations that we select, recording the development and assisting exploration of developments (permitting the undoing of past actions, retracing of steps, etc). We use the POPART system [Wile 81a] to provide this support, and intend to use the PADDLE language [Wile 81b] to record developments.

There remains the problem of how to decide what order to consider mapping the instances of constructs in a sizeable specification. We may expect that the nature of an individual specification will significantly affect this ordering, rather than using a preformulated ordering based on comparison of the Gist constructs in the abstract. Since our experience with this issue is rather limited, this remains an area where we will rely heavily upon human intuition for guidance.

Through adopting the transformational approach, advocated by many groups (e.g. [Balzer, Goldman & Wile 76], [Bauer 76], [Darlington & Burstall 76]), we hope to inherit the claimed-for advantages and already discovered techniques of this approach. Since our objective is to develop a prototype with tolerable efficiency for testing purposes, as opposed to a polished implementation with near optimal efficiency, we may expect to have an easier task of development. Furthermore, we note that in our view a specification may denote a set of possible behaviours, and that an implementation need exhibit only a subset[1] of those behaviours. Hence during development of a prototype we may, for expediency of development, implement a subset of the behaviours denoted by the specification, provided that we remain aware that testing of such a prototype will not reveal all the possible behaviours of the original specification.

Our approach to specification is to describe a closed world, within which lies the portion we are to implement. Because this portion may have only limited access to, and control of, the remainder of that world, it may be impossible to implement this portion so as to achieve the behaviours required of the world whilst conforming to these limitations. We may use prototyping to prove (by construction) that an implementation is possible. Certainly if our prototype is to be tested by plugging it in to an immutable environment, then in our development we must conform to all the restrictions of that interface. If, however, we have latitude to make changes to that environment for our primary purpose of observing implied behaviours, then we may make use of such freedoms in developing our prototype. Of course, if we do so, we should separately convince ourselves that implementation conforming to all the restrictions will later be possible.

2. Mappings

In this section we consider Gist's high level specification constructs and present some implementation options, selection criteria and mappings for each. The mappings for a construct fall into one of two classes:

- generally applicable techniques, and
- specific techniques for commonly occurring idioms of usage of that construct.

As we investigate more examples, we will identify the commonly occurring idioms, and build up a collection of mappings to deal with them. When faced with an instance which none of these mappings can handle, we may

- apply a general technique,
- compose a sequence of low-level transformations to deal with that particular case, with

[1] a non-empty subset, assuming the specification denotes a non-empty set

the option of abstracting that composed sequence to form a new mapping for our collection, or

make some change by hand and verify the correctness of that change (again with the option of abstracting this into a new transformation).

Techniques for forming new transformations via verification or composition of existing ones have been suggested by the CIP group, e.g. [Broy & Pepper 80]. Collections of transformations are a familiar notion, e.g. [Standish et al 76].

Gist's constructs, in the order we consider them, are *historical reference*, *constraints and non-determinism*, *derived information*, *demons* and *perfect knowledge*. Historical reference is presented first as a self contained and relatively straightforward construct to deal with.

A knowledge of the constructs and their utility for specification is presupposed, and may be found in the companion paper, [Balzer et al 82], along with a specification of the package router problem, from which we draw illustrative examples. A brief description of this problem may be found in the appendix.

2.1. Historical reference

The computational expense we wish to avoid is one of retaining the entire history of states in order to answer any potential future query.

2.1.1. Implementation options

One implementation option for historical reference is to seek to derive the required information from information that is present in the current state, then the historical reference can be replaced by that derivation. In general, however, it will be necessary to introduce and maintain auxiliary data structures to hold information that might be referenced at some time in the future; such references can be modified into retrievals of the information from these introduced structures.

Having eliminated all historical references from a specification, an interpreter need no longer remember past execution states.

2.1.2. Selection criteria

The tradeoffs between alternative options are the classical space/time tradeoffs arising from comparing cost of derivation with cost of storage and maintenance of redundant information to permit simple retrieval. Since our objective is rapid development of a tolerably efficient prototype, we may be ready to tolerate some less than optimal choices which ease our task of transformation. Our experience suggests that specifications contain a relatively small number of simple historical references, and in most cases we would be satisfied with a relatively crude mapping from the efficiency standpoint.

2.1.3. Mappings

The mapping to re-express a retrieval in terms of existing information relies essentially on the demonstration of equivalence of those pieces of information. The transformation we apply to do this mapping is the simple one permitting the replacement of any expression with any equivalent expression. Hence to perform such a transformation we must first compose an expression in terms of present information only (currently an activity we must do entirely by hand), and then demonstrate that it is equivalent to the historical reference. The latter requires some form of theorem proving activity; we are in the process of constructing analysis tools for Gist that will enable us to answer simple queries, and form the basis of a theorem prover. We retain the option of accepting advice from the human developer in terms of an unproven (at least by our system) lemma, which can be used to permit application of some transformation step. The result of the development activity would then be a transformed specification, together with the development record, and a set of lemmas, upon whose correctness the correctness of the development depends.

2.1.4. Examples

Mapping through introduction and maintenance of auxiliary data:

When a new package arrives at the source, it is necessary to know the destination of the immediately preceding package to have been there. We may recognise this as an instance of the idiom of referring to the time-ordered sequence of objects satisfying some predicate, here the objects being packages, and the predicate being located at the source. Our mapping for this idiom introduces explicit maintenance for the sequence of packages to have been located at the source (in order of their arrival there), and re-expresses the historical reference as a retrieval of the destination of the package immediately preceding the last package in that maintained sequence. For the purposes of proptotyping this might be sufficiently efficient, although clearly we can improve it further by maintaining only the last two elements of the sequence.

Mapping by derivation in terms of current information:

In the package router, we may wish to know[2] for a package within the network which was the last switch it went through. This could be derived from knowledge of the package's current location, and of the static topology of the network.

---

[2] the actual specification makes no such demand, this is posed for purposes of illustration only.

2.2. Constraints and non-determinism

The combination of constraints and non-determinism serves as an extremely powerful and convenient specification mechanism. Naive interpretation - by exploring the choice paths in a depth first manner, with backtracking on detection of constraint violation, would in general be computationally infeasible.

2.2.1. Implementation options

There exists a range of possibilities between two extremes for mapping away constraints and non-determinism.

> At one extreme we may seek a purely "predictive" solution, in which the non-deterministic choices are refined to become selections of just those choices guaranteed not to violate the constraints at any time in the future. Having made this choice the constraints are then redundant, and may be discarded.
>
> At the other extreme we seek a "backtracking" solution, with the choice points as backtracking points, and the constraints mapped into code to check for violation of constraints, and cause backtracking if such violation is detected. In non-AI programs such search is normally coded as an iterative loop through a space of possibilities filtering out the inadmissible ones.

Since a "predictive" solution will not always be readily obtainable (or even possible), we may retain some backtracking in our developed prototype. This can nevertheless be made significantly more efficient than the original specification, since at least some of the constraints might be eliminated by refining choices, heuristics could be introduced to order selection of choices to pick first those most likely to be successful, and the checking of remaining constraints refined from every state change to just those places where violation could potentially occur.

The backtracking solution may imply inserting code into the environment outside of our portion, and/or undoing actions of the environment. We may only do this if, for prototyping purposes, we may alter the environment too, and must remember that when it comes to actual implementation some alternative solution will be required. In general it may be impossible to implement only a portion of a system in such a way as to guarantee satisfaction of all the constraints over the whole system.

2.2.2. Selection criteria

Our preference is towards "predictive" solutions, since they avoid the need for backtracking. As with historical reference, our search for an optimal solution is balanced by the need for rapid development.

Our experience suggests that resolution of constraints and non-determinism is one of the key development steps in determining the nature of the resulting implementation, and hence we believe it should be one of the earlier steps in a development.

2.2.3. Mappings

A generally applicable mapping is to unfold the constraints into explicit checks, with backtracking if violation is detected, and apply transformations to move the checks back (with respect to the time at which they will be executed) toward the choice points. Explicit checks need be introduced only after those points of transition that could potentially lead to violation of the constraint's predicate. Identification of such points is an instance of a more general identification problem, which arises not only in unfolding constraints, but also maintaining derived information and unfolding the triggering of demons. We defer discussion of this until the mappings section for derived information, 2.3.3. Once constraint checking has been made explicit, we may apply low-level transformations to move these checks and backtracks through the code, in order to bring them nearer the non-deterministic choice points, and ultimately merge the requirements into the selections. If we assimilate all the constraint checks into the choice points, we will have succeeded in achieving a predictive solution; otherwise some implementation of backtracking remains.

2.2.4. Examples

Predictive solution:

In the package router, switch setting is defined in a very non-deterministic fashion (at random times, a switch sets itself to any one of its outlets). This non-determinism is constrained by (i) requiring that the switch be empty when the setting takes place, and (ii) by prohibiting a state in which a package reaches a switch set improperly for that package when there had been some past opportunity to set the switch correctly for that package. This combination of non-determinism and constraints serves to denote behaviours in which switches are set in advance (when possible) so as to route packages correctly.

The requirement that a switch be empty when setting takes place can be moved from the precondition of doing the setting into the non-deterministic choice of when to do setting, refining random to random when the switch is empty. This is achieved by recognising switch setting as an instance of a demon with a random trigger (a common idiom for expressing non-determinism), whose response begins with a requirement; for this we have a transformation to conjoin the predicate in the trigger with the requirement's predicate, and delete the requirement from the response, in effect refining the triggering to just those occasions which will not cause violation of that precondition[3].

---

[3] Actually, to apply this transformation we must also check that the event of that demon triggering is not observed anywhere, and ensure that our modified demon will begin performing its response before anything else happens.

The prohibition (ii) is dealt with by recognising it as an instance of the idiom

always prohibited P_1 and (P_2 asof ever)

where P_1 and P_2 are predicates[4], P_1 corresponding to the identification of a package in an improperly set switch, P_2 corresponding to the identification of an opportunity to set the switch. The mapping for this idiom suggests that on P_2 becoming true, the choice is made to ensure that P_1 will remain false, i.e. on there being an opportunity to set a switch for a package, ensure that the package never reaches that switch improperly set, which we refine further into immediately setting the switch correctly (and leaving it set that way until the package has passed or is diverted higher up from reaching the switch).

Backtracking solution:

The eight queens problem (of placing 8 queens on a chessboard in such a way as to not have any queen attacking any other queen) is a good example of a problem where backtracking remains in the final solution, yet some of the constraint checking gets incorporated into the selections (e.g. having placed one queen on a row, don't even try to place another queen on that same row, as it is sure to violate the no-capture requirement). See [Balzer 81] for a detailed description of a development of a solution to the 8-queens problem that illustrates this and other aspects of mappings.

Impossible to achieve a solution:

If the package router required that packages always get routed correctly, whilst retaining the restrictions that our portion may not control either the rate of movement of packages or the length of the delay periods before their release into the network, then no solution would be possible.

2.3. Derived information

Derived information is information derived from other information. Only in instances where frequent reference is being made to information with a complex derivation is it necessary to worry unduly about careful strategies for implementing the derivations. We may expect a default strategy, of performing the entire calculation each time the information is referenced, to be satisfactory in most cases.

[4] (P_2 asof ever) is a predicate which is true if P_2 is true in the present or any preceding state.

2.3.1. Implementation options

An alternative to the "backward inference" approach of unfolding the derivation at the site of reference, is the "forward inference" approach in which the derived information is continually maintained. At any point where a change takes place that might potentially change the value of some derived information, code is introduced to recalculate and store the changed value. Retrievals of the derived information become retrievals from the data structures introduced to store the maintained values.

Additionally, in updating the value of (changed) derived information, it may be possible to take advantage of knowledge of its old value, and incrementally compute the new value rather than having to perform an entire recomputation.

Backward and forward inference are at opposite ends of a spectrum of options, interior points being the forward maintenance of part of the information, and backward computation of the remainder on reference.

2.3.2. Selection criteria

Our selection criteria are hence based on our judgement on the frequency of access and cost of recomputation on each such access vs the expense of auxiliary storage and the cost of repeated updates to maintain the stored information.

2.3.3. Mappings

Backward inference involves simply unfolding the derivation at the sites of reference (or leaving the references as function calls to the derivation).

Forward inference depends upon identifying points of transition where a change takes place which might potentially change the value of the derived information, and inserting code to perform recomputation and storage of the changed value. References to the derived information are modified to be retrievals from the stored information. This assumes that all the points where such code are to be introduced fall within our portion. If not, such introduction of code into the environment is predicated upon the freedom to modify that environment for prototyping purposes (with the consequent realisation that some alternative mapping must be applied in actual implementation).

The problem of identifying points of transition that could potentially change the derived information is an identification problem common to mappings for several constructs. In the mapping to unfold constraints we saw the need to identify transitions that might lead to violation of a constraint's

predicate; as we shall see in the section on demons, there is the need to identify transitions that might lead to a demon's trigger predicate changing in value from false to true. For derived information, the change could be to some arbitrary expression (not necessarily a predicate). In each case, code is to be inserted at each point to test whether the change has indeed taken place, and if so to cause the appropriate response (for constraint violation, backtracking; for demon triggering, the demon's response; for derived information, storage of the new value).

Maintenance of derived information is further complicated by seeking to take advantage of knowing the information's value prior to the transition, in order to incrementally compute the new value rather than perform an entire recomputation.

The overall technique of incremental maintenance of information has been extensively explored in the SETL framework, and is often called *formal differentiation* or *finite differencing*. See [Koenig & Paige 81] for details and pointers to further work. Their findings are readily adaptable to the Gist framework, and should prove to be of significant utility.

2.3.4. Examples

Incremental maintenance: During development of the package router prototype, we find frequent reference being made to the sequence of not-yet-misrouted packages heading towards a switch. Since this appears a relatively complex item of derived information, we judge it worthwhile to maintain incrementally. This leads to introduction of code in several places, for example, where a package is observed to appear at the source; here the code appends that package to the end of each of the sequences of packages associated with switches on route to its destination. Appending the new package to the end of a sequence is a particularly efficient incremental calculation of the new value.

2.4. Demons

Demons are data-invoked processes, their semantics being that upon any transition that causes the trigger of a demon to change in value from false to true, that demon's response is invoked. The computational expense we wish to avoid is that of a naive interpretation in which all the demon's trigger predicates are examined after every transition.

2.4.1. Implementation options

For prototyping, we may apply the general technique of unfolding checks for demon triggering into explicit checks into those transition points which can be identified as potential triggering points.

As with derived information, this may imply chagnes beyond the boundaries of our portion. Instead we might have to resort to, say, implementing polling loops to watch for transitions caused by the environment which trigger our demons.

2.4.2. Selection criteria

In the event of an instance of a demon matching one of the idioms for which we have a transformation, we would use that, otherwise fall back on the general technique of unfolding.

2.4.3. Mappings

Identifying potential triggering points is an instance of the identification problem described earlier in section 2.3.3.

2.4.4. Examples

Unfolding a demon:

One of the demons in the package router signals arrival of a misrouted arrival in a wrong bin. Unfolding this demon leads to code introduced into the package movement portion, which, on moving a package into a bin, checks whether that bin is the package's destination, and if not causes the appropriate signal.

2.5. Perfect information

Perfect information during specification makes the assumption that all information about the world is always available, whereas an implementation may have access to only limited information about its environment. This is another instance of implementation restrictions that we may choose to ignore for prototyping purposes, given the freedom to alter the environment (with the usual caveat that if we do so, we should separately convince ourselves that a conforming implementation will be achievable).

2.5.1. Options, selection criteria and mappings

Reliance on perfect information is very similar to reliance on historical information. It may be possible to derive the required information from available information, or it may be necessary to introduce and maintain auxiliary data structures to hold information that might be required to make the derivations. We apply the same selection criteria as for historical reference options, and share the same style of mapping techniques.

2.5.2. Examples

The package router places a limit on our knowledge about packages once they are inside the routing network. In particular, we cannot determine their destinations, nor may we know even their identities once they have been released from the source. Our only information is from sensors on

entries and exits of switches, and entries of bins, that indicate when a package has passed. The specification makes widespread use of knowledge of package identities and destinations. By introducing and maintaining our own queues to store the identities and destinations of packages at each location within the network, we may derive the required information from these queues. Observation of sensor triggering is used to maintain the correspondence between the packages in the network and the information in our queues.

Insufficient information to achieve an implementation:
Clearly the limited information that sensors provide is necessary to perform routing. If, say, we had omitted to provide sensors on entries to bins, it would prove impossible to detect when a misrouted package had actually reached the bin, and hence impossible to make the required signal of that event.

2.6. Data representation

We have not explored in detail the problem of selecting appropriate data structures with which to represent our relational information; we believe this to be less crucial to the development of testable prototypes than dealing with Gist's major specification constructs. In our application of this technology to implementation we believe the same framework - of techniques and idiomatic transformations - is suited to dealing with instances of relations. We intend to adapt the results of other researchers in this area, e.g. that of the SETL group, [Schonberg, Schwartz & Sharir 81].

3. Summary and Conclusions

We have proposed using a methodology for developing testable prototypes from specifications, based upon instances of specification constructs as the focal points of the development. We believe the overall approach is suitable for any specification language that uses constructs tailored for expressiveness rather than efficiency. For each of Gist's major constructs we have discussed mappings to implement them in a reasonably efficient manner.

More details on the mappings work, with the emphasis on development of a polished implementation rather than a testable prototype, may be found in [London & Feather 82]. Our experience confirms our belief that prototyping is easier than developing a polished implementation, taking advantage of both the lack of necessity for optimal efficiency and freedoms not available to an actual implementation (e.g. use of perfect information).

The methodology allows the incremental incorporation of techniques (gathered by adapting those of other researchers, and by discovering new ones ourselves) and provides a convenient framework within which to introduce machine support for the various activities.

Much work remains to be done, in particular, building an analyzer and theorem prover, and experimenting with more examples, in so doing accumulating a catalogue of transformations for commonly occuring idioms. We need to investigate the issues of feedback from the testing of a prototype to suggest modifications of the specification, and of making use of the effort already expended (to develop a prototype) when it comes to developing a prototype of a slightly modified specification, or developing a polished implementation.

Acknowledgements

The author wishes to thank Phil London, with whom the mappings research has been performed, and the other members of the ISI Transformational Implementation group: Bob Balzer, Wellington Chiu, Don Cohen, Lee Erman, Steve Fickas, Neil Goldman, Bill Swartout, and Dave Wile. They collectively have defined the context within which this work lies, and individually have improved this paper through frequent discussions about the research and helpful comments on the drafts.

Bibliography

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

[Balzer et al 82] Balzer, R., Goldman, N. & Wile, D., "Operational specification as the basis for rapid prototyping," in *Proceedings, ACM SIGSOFT Software Engineering Symposium on Rapid Prototyping*, 1982.

[Balzer, Goldman & Wile 76] Balzer, R., Goldman, N. & Wile, D., "On the transformational implementation approach to programming," in *Proceedings, 2nd International Conference on Software Engineering, SanFransisco, California*, pp. 337-344, October 1976.

[Bauer 76] Bauer, F.L., "Programming as an evolutionary process," in *Proceedings, Second International Conference on Software Engineering, San Fransisco, California*, pp. 223-234, IEEE, 1976.

[Bauer et al 78] Bauer, F.L., Broy, M., Gnatz, R., Partsch, H., Pepper, P. & Wossner, H., "Towards a wide spectrum language to support program specification and program development," *SIGPLAN Notices* 13, (12), December 1978, 15-23.

[Broy & Pepper 80] Broy, M. & Pepper, P., "Program development as a formal activity," *IEEE Transactions on Software Engineering* SE-7, (1), 1980, 14-22.

[Cohen et al 82] Cohen, D., Swartout, W. & Balzer, R., "Using symbolic execution to characterize behavior," in *Proceedings, ACM SIGSOFT Software Engineering Symposium on Rapid Prototyping*, 1982.

[Darlington & Burstall 76] Darlington, J. & Burstall, R.M., "A system which automatically improves programs," *Acta Informatica* 6, (1), March 1976, 41-60.

[Koenig & Paige 81] Koenig, S. & Paige, R., "A transformational framework for the automatic control of derived data," in *Proceedings, 7th International Conference on Very Large Data Bases*, pp. 306-318, September 1981.

[London & Feather 82] London, P. & Feather, M.S., *Implementing specification freedoms*, ISI, 4676 Admiralty Way, Marina del Rey, CA 90291, Technical Report RR-81-100, 1982. Submitted to Science of Computer Programming

[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.

[Standish et al 76] Standish, T.A., Harriman, D.C., Kibler, D.F. & Neighbors, J.M., "Improving and refining programs by program manipulation," in *ACM National Conference Proceedings*, pp. 509-516, ACM, 1976.

[Wile 81a] Wile, D. S., *POPART: Producer of Parsers and Related Tools, System Builders' Manual*, ISI, 4676 Admiralty Way, Marina del Rey, CA, 90291, 1981.

[Wile 81b] Wile, D. S., *Program Developments as Formal Objects*, ISI, 4676 Admiralty Way, Marina del Rey, CA 90291, Technical Report RR-81-99, 1981.

I. Package Router Problem

To illustrate our approach, we choose as an example a routing system for distributing packages into destination bins. Figure 3-1 illustrates the routing network. At the top is a source station which 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.

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; so 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 (pre-calculated) fixed length of time (to reduce the chance of bunching). In spite of such precautions, packages may still bunch up and become mis-

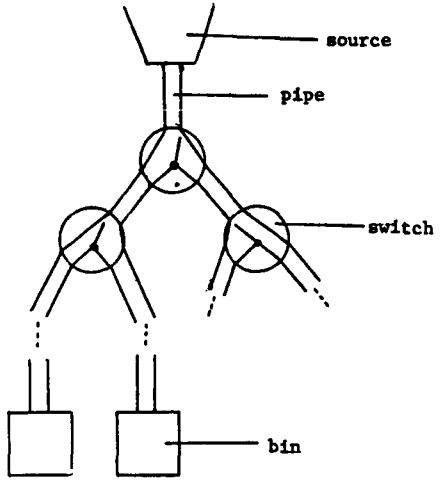

Figure 3-1: package router

routed, 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 behavior. 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 are sensors placed on switches and bins; these detect the entry and exit of packages but are unable to determine their identity. (The sensors will be able to recognize the passage of individual packages, regardless of bunching).