AD-A127 330 TESTING OF CONCURRENT PROGRAMS AND PARTIAL SPECIFICATIONS(U) MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE D HAMLET DEC 82 TR-82/13 1/1

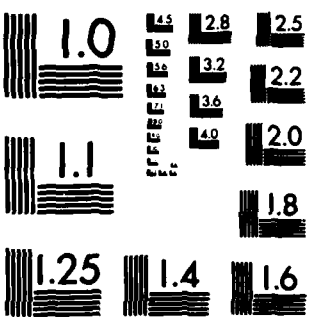
UNCLASSIFIED AFOSR-TR-83-0306 F49620-80-C-0001 F/G 9/2 NL

END

DATE FILMED

5 83

DTIC

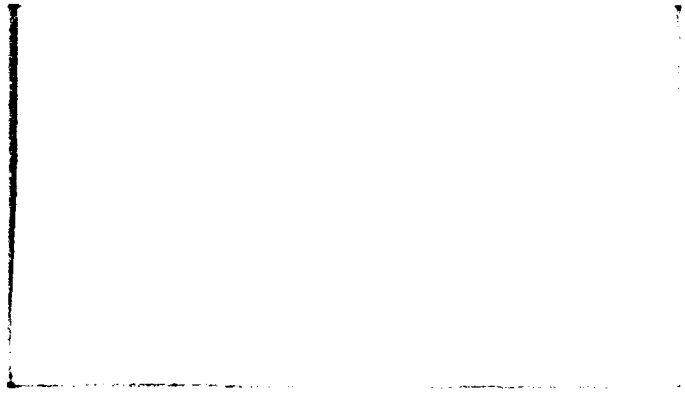


MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AFOSR-TR- 83-0306

(12)

AD A127330



DTIC
ELECTE
APR 28 1983
S
A

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF MELBOURNE

DTIC FILE COPY

Approved for public release;
distribution unlimited.

83 04 27 018

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DTIC
This technical report has been reviewed and is
approved for public release IAW AFR 190-12.
Distribution is unlimited.
MATTHEW J. KELLY
Chief, Technical Information Division

TESTING OF CONCURRENT PROGRAMS AND PARTIAL SPECIFICATIONS

Dick Hamlet

Technical Report 82/13
December, 1982

Department of Computer Science
University of Melbourne+
Parkville, Victoria 3052
Australia

DTIC
ELECTE
APR 2 3 1983
A

Abstract

The testing problems of concurrent systems include those of sequential programs, but there are two additional difficulties: (1) the scheduling of tasks may alter the behavior, making tests misleading; (2) testing may be conducted at an early stage of development, by users who are not software experts. Concurrent-process systems can be modeled by a collection of finite-state transducers, in a way that displays their unique problems. The specification languages PAISLey and Gist approach the definition of concurrent systems differently, but both permit users to execute partially defined systems. The declarative language PROLOG, although not explicitly designed for concurrent programming, exhibits similar characteristics. Prototype execution has some unexpected implications for testing, and for final implementation.

+On leave from Department of Computer Science, University of Maryland, College Park 20742, U S A. This work was partially supported by USAFOSR grant F49620-80-C-[redacted].

0001-a-

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR- 83-0306	2. GOVT ACCESSION NO.: AD-A127330
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): TESTING OF CONCURRENT PROGRAMS AND PARTIAL SPECIFICATIONS	5. TYPE OF REPORT & PERIOD COVERED: TECHNICAL
	6. PERFORMING ORG. REPORT NUMBER: TR #82/13
7. AUTHOR(s): Dick Hamlet	8. CONTRACT OR GRANT NUMBER(s): F49620-80-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Computer Science, University of Maryland, College Park MD 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: PE61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Mathematical & Information Sciences Directorate, Air Force Office of Scientific Research, Bolling AFB DC 20332	12. REPORT DATE: DEC 82
	13. NUMBER OF PAGES: 12
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

18. SUPPLEMENTARY NOTES

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

DTIC COPY INSPECTED

Distribution/ Availability Codes Dist Avail and/or Special

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

The testing problems of concurrent systems include those of sequential programs, but there are two additional difficulties: (1) the scheduling of tasks may alter the behavior, making tests misleading; (2) testing may be conducted at an early stage of development, by users who are not software experts. Concurrent-process systems can be modeled by a collection of finite-state transducers, in a way that displays their unique problems. The specification languages PAISLey and Gist approach the definition of concurrent systems differently, but both permit users to execute partially defined systems. The (CONTINUED)

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

83 04 27 018

ITEM #20, CONTINUED: declarative language PROLOG, although not explicitly designed for concurrent programming, exhibits similar characteristics. Prototype execution has some unexpected implications for testing, and for final implementation.

1. Software Implemented for Parallel Processing

Testing of any software faces the problem of insufficient data--the cases used may have failed to expose a fault that will be excited later in some untried case. For concurrent systems there is an additional problem posed by scheduling of the cooperating processes: test success for one case may be an accident resulting from a lucky interleaving of process actions.

Testing has assumed a new aspect with the appearance of executable specifications. Specification languages are intended to keep the end user informed about a system under development, by allowing test execution early in the requirements/specification stage. If the results of these tests are misleading, both the user and system developer may be lulled into a belief that faulty specifications are correct. Execution of unfinished specifications makes it more likely that the test results will be misinterpreted.

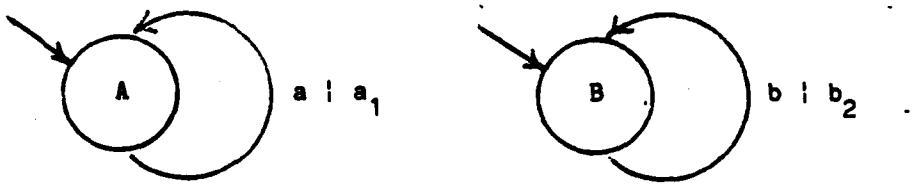
After presenting a simple model of parallel computation, this paper gives brief descriptions of several executable specification languages, and considers their testing problems.

2. A Simple Model of Parallel Computation

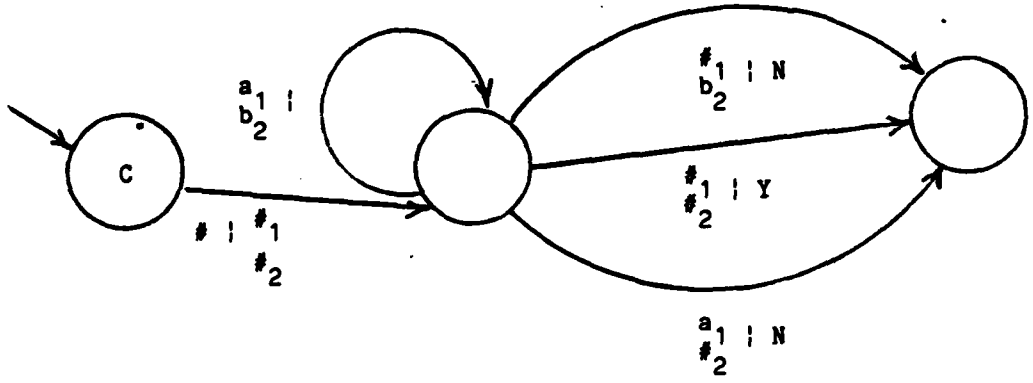
Languages for parallel programming and specification of systems to be implemented as cooperating processes are necessarily complex and powerful. But the essential difficulties of testing such programs can be understood in terms of a model in which each process is a named nondeterministic finite-state transducer with empty moves, modified to communicate with other processes. To this end a transducer (process) may change state on the basis of symbols obtained from other transducers, and may output to other transducers. This communication takes place as follows: any interprocess output produced is appended to one of a finite number of labeled communications strings, and interprocess inputs are obtained from these strings by name. The difference between a collection of such transducers and conventional ones is that the input (and communications strings) are thought of as dynamic objects. The transducers are imagined to be active in parallel, with their transitions occurring asynchronously. Should any process need a symbol when the necessary string is empty, it waits until there is a symbol available. Any of the several processes can consume a symbol that appears on the input or communication strings. Where more than one possibility exists for the disposition of a symbol, all the choices may occur. (However, each transition is completed before another begins, so the entire system operates in a series of discrete steps, without any real simultaneity.)

For example, the collection of three process {A, B, C} shown below will respond to strings of 'a' and 'b' symbols before '#' and output 'Y' if there are the same number of each, or 'N' if there are not. Inputs are shown to the left of the bar, and where not all possibilities are listed, empty input is implied. To the right of the bar are outputs, which may also be empty. The subscripts refer to distinct communications strings; external input and output

are not subscripted.



Processes A and B copy symbols (respectively) onto communications strings 1 and 2. Process C:



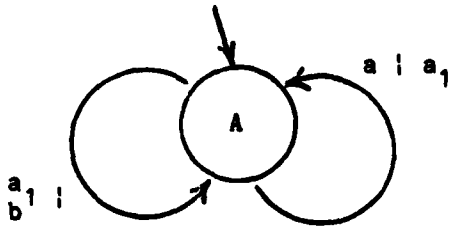
then judges the pair of communications strings and produces the proper result. The computation is controlled in that it is driven by the input--each symbol can activate only one of the processes--but when characters follow the '#', processes A and B continue to add to the communications strings, although this does not alter the output.

This language isn't very like actual specification languages, but it exhibits the features necessary for a discussion of testing. In it, programs can be written so that they cooperate in a perfectly deterministic way, synchronized so that the pattern of which process is active is controlled at all times. It is also possible to write programs with more chaotic patterns of cooperation, including the possibility of "deadlock." Deadlock is defined as a situation in which all processes are in an interprocess input state, and the necessary symbols are not present. This situation will then persist. The cases in between are the ones of interest: the pattern of process activation has many possibilities, but none of these lead to deadlock, and the overall input-output behavior of the system is the specification. In the example above, the computation is synchronized until '#' is seen in the input, then there are several ways that the processes can be interleaved. However, the output does not depend on which sequence occurs; indeed, if input following the '#' is slow in coming, the output appears before its arrival.

Such operational specifications are not necessarily functional (as the example is), in the sense that for a single input string there is at most one output-string possibility. It can happen that there is competition for an input or communications-string symbol, and depending on which transition is taken, the outcome differs. A system is <u>consistent</u> if this cannot happen; that is, for each input, the same output must appear no matter which choices are made about dispersing input or communications-string symbols. For a given input, consistency can be checked by trying all the sequences, but in practice

the task is combinatorically intractable.

To illustrate nonfunctional behavior, consider the example above, but with a slight change intended to make it more efficient. Instead of collecting all the input symbols on two strings, add to process A an additional transition that matches and discards adjacent pairs, so that it becomes:



but processes B and C remain the same. The new specification has some behaviors that are the same as those of the original, because the transition added to A is not chosen in preference to the original one in B; it has some new behaviors that lead to the desired result in the desired way; but, it has some perhaps unexpected behaviors that lead to inconsistent results. For example, on input

aabb#b

all three situations can occur, the last because the extraneous symbol causes the new transition in A to be taken in preference to the matching operation in C, with the result that the two communication strings do not balance.

It is always possible to control undesirable behavior by incorporating a kind of scheduling control into the system being designed: one communications string can be used to record the sequence of activations, and a process can examine it for correctness. As a final element in our model, this idea is given a convenient expression. Imagine that as the processes are activated, a "control string" records the sequence of process names. For example, in the system with multiple behaviors described above, the control string might contain:

AABBCCBCC (output Y)

AACBCC (output Y)

AABBCACC (output N)

(among others) for input aabb#b , corresponding to the cases described above. Along with the processes, our model then allows a regular expression to constrain the contents of the control string. A control string not in the language described by this "control expression" cannot occur. In the example, a control expression of

$$(A \mid B)^{\#}C^{\#}(A \mid B)^{\#}$$

would confine operation to the desired cases. (The control expression is not an accurate description of the actual possibilities; it merely omits those we want to reject.)

3. Executable specification languages

In the conventional software life cycle, requirements are given to a professional organization where technical development takes place, later culminating in software delivery to the end user. Because the communication between users and software professionals is poor, and because technical decisions alter the software's capabilities, it is commonplace to find that although the developer believes requirements are satisfied, the user disagrees. Evidently the only way to attack this problem is to make requirements more precise, with less room for differing interpretations. Professionals naturally use programming languages, with their carefully defined syntax and semantics, as a model for what a requirements/specification language should be. Precision in stating requirements should benefit the software developer independent of communication with users, because formal languages sharpen ideas and encourage analysis. However, from the user's side, the direct advantage of formal requirements is less clear. The user has no expertise save in his area of application, which will be only dimly reflected in a formal language, so he will still have to deal with computer experts in unfamiliar terms.

Two characteristics of formal requirements languages can help reduce the odds against the user. If a specification is executable even when unfinished, it can be used as a "prototype" with which a user can experiment. As each technical decision is made, the user can see it reflected in a prototype, and express immediate approval or disapproval. Tests of complex systems are not a panacea, even when the systems are completely implemented and considerable thought given to the test plan. A user cannot be expected to devise and monitor anything like an acceptance test of requirements, particularly with a changing prototype. It is therefore important to investigate the properties of such tests.

This section briefly describes three languages now under development, relates them to the concurrent-process model of Section 2, and describes the execution of an unfinished specification. Each of these languages has many features not described here, features which are advantageous to the writer and reader of specifications, but irrelevant to the problems of testing.

3.1 PAISLey

A PAISLey (Process-oriented, Applicative, Interpretable Specification Language) specification [1] is a fixed collection of processes, each constrained to act in a cyclic fashion, communicating through named message channels. Each process is described as a function mapping a data type onto itself, and one cycle of operation corresponds to the evaluation of the process function for some input, resulting in an output which becomes input to the next cycle. Process functions are defined using three devices: (1) auxiliary functions, defined in the same way as process functions, but without autonomous existence; (2) McCarthy conditionals in which a sequence of tests is used to select a value; and (3) "exchange functions" for interprocess communication. Since each of these devices has a functional appearance, they are combined using composition alone. Auxiliary functions are thus like typed pure-LISP functions (except for the exchange functions); process functions are the same, but are invoked repeatedly using each result value as the next input.

PAISLey processes interact only through the evaluation of exchange functions. Each such function names a channel, and when functions with the same channel name are evaluated in two processes, they may synchronize and exchange parameters. The synchronization may be required, so that evaluation

of the exchange function is not complete unless two processes do match channels; or, it may be allowed to fail. Using these two mechanisms, processes can be forced to synchronize, or can continue after a failed attempt. Constraints on process execution may be supplied in the form of timing information. Any function may be given (for example) a minimum or maximum time for evaluation.

The collection of processes that is a PAISLey specification is imagined to behave in consort. The function evaluations within each process are performed independently, except where exchange functions force an ordering. Timing constraints can make synchronization impossible, or eliminate otherwise-possible orderings. For example, a process that is required to complete its cycle in less than 1 second cannot synchronize once each cycle with another that is required to take longer than 1 second, nor can it use an auxiliary function with a 2-second evaluation time. There are in general many orderings of the independent process cycles that meet the timing and synchronization constraints of the PAISLey specification. The collection of these proper behaviors is the meaning of the specification.

An unfinished PAISLey specification may have missing processes, and hence exchange functions in other processes may not be matched. Or, auxiliary function definitions may be missing. Both situations are part of the specification methodology: missing processes correspond to sections of the specification that are not yet developed; missing functions correspond to refinements to be made later in the usual top-down way. A missing auxiliary function and an exchange function whose matching channel is in a missing process both appear as undefined functions. Finally, timing constraints may be missing or weaker than they should be, permitting some behavior patterns that were meant to be excluded.

Execution of a PAISLey specification [2] requires as "input" those auxiliary-function values that are undefined, and similarly values for any unsatisfied exchanges. Because these inputs are coupled with "output" that is the corresponding parameter value, it is convenient to supply them interactively, and with prior knowledge of the output. Then the execution of processes may be completely determined by the required synchronizations and timing constraints. However, the PAISLey execution system does not attempt to determine if this is so, but rather chooses random timings that meet the constraints, leaving undetermined scheduling choices to the human user, who also supplies the inputs. If timing constraints are violated during execution, the test is terminated. There is no backtracking to attempt to determine if different random timing choices, or different scheduling decisions, would have satisfied the constraints. Thus the direct execution of a specification can fail when there exist behaviors that would meet the specification. On the other hand, when direct execution of a specification succeeds, there may exist other ways to meet the specification with different behavior. The latter is particularly likely if the user had to make many scheduling decisions, since the timing characteristics of the system remain relatively underspecified. As recompense for these defects of direct execution, it is efficient and easy for an inexpert user to control.

The finite-state model of Section 2 is conceptually close to a PAISLey specification, except that the power of individual processes may be beyond that of the finite-state, and the interprocess communication channels are more flexible than the communication strings taken there. The cyclic nature of processes can be captured by insisting that the finite-state transitions include proper loops, and the possibility of a bypassed synchronization is modeled by nondeterministic machine transitions. Some PAISLey timing constraints can be modeled by the control expression, but not in a very natural way: the times must be converted into the patterns they establish.

3.2 Gist

A Gist specification [3] has roughly the following components:

Types, similar to those in a strongly typed programming language.

Relations, similar to boolean procedures of a conventional programming language, with parameters of the declared types. Quantifiers may appear in relations, with range the collection of objects created by actions.

Actions, which either create or delete objects (of declared types) or relationships (assertions that relations hold). These may be combined in conventional ways, by sequence, iteration, etc.

Demons, processes activated by a trigger relation. The demon includes actions; after they are performed the process is deactivated.

Constraints, assertions that relations always (or never) hold.

The meaning of a Gist specification is that demons' actions are performed whenever the triggering relation holds, giving rise to collections of possible actions that arise from different time sequences (should more than one trigger at the same time). The intent is that the proper system behavior is among the possibilities. However, many of the sequences are unwanted, and these are to be eliminated by the constraints. When a sequence violates a constraint, it is not part of the specification meaning.

An unfinished Gist specification may have missing demons or actions missing within a demon (so that intended actions are not performed); or, missing or too-weak constraints (so that some undesirable action sequences are not eliminated). Gist favors specification construction by progressive elimination of the latter: beginning with action sequences that are too broad, constraints are added to successively refine the behavior.

A Gist specification can be executed directly [4]. Given an initial universe (which may be empty if the specification has an "input" portion), the execution is a simulation whose events are the demons triggered by the state of the universe, and whose actions transform that state. The possibility of many demons triggering at once, with different successive states depending on the sequence of their actions, provides the leeway for constraints to reject some simulations. A Gist interpreter therefore simulates one of the action sequences, using any ordering of demons, until a constraint is violated, then backtracks to try a different order. This execution can be very inefficient if there are many sequences and most are eliminated. However, of more concern is the case of a too-easy success: if the simulation chosen happens to be one among many that differ widely, execution can mislead the user into thinking that an unfinished specification is adequate. The effect is particularly pernicious if the user controls execution by selecting demon ordering, since then he may unconsciously select the sequences he hopes to have specified.

Although the actions of Gist demons do not correspond very well to finite-state behavior, and communication among demons uses a global state that is not captured well by communication strings, the simple model of Section 2 roughly describes the interplay of demons' actions. The model does even less well with Gist constraints. Its control sequences have the ability to forbid behaviors as constraints do, but the latter are in a declarative rather than an operational style and may be expressed in terms of the state as it evolves.

Despite the power of Gist constraints, they cannot express such properties of a specification as consistency. These must be proved outside the language, because the quantification in constraints is over objects of one input universe, not over all universes. In such metaproofs, the constraints play the major role, however. A proof of consistency could be attempted by a theorem prover, working in a first-order system whose objects included input universes, and using the Gist constraints as axioms.

3.3 PROLOG

PROLOG is usually considered to be a programming language instead of a specification language; it makes no explicit provisions for concurrent processing. However, its declarative form, and the need to produce efficient versions of its "programs," are closely connected to the intuitive idea of specification. Execution and testing of PROLOG programs illuminates the prototype aspect of operational languages.

A PROLOG program consists of an unordered collection of first-order logic formulas, each in the restricted form of a universally-quantified implication with conjunctive antecedent and disjunctive consequent [5]. The program may be thought of as a set of axioms, to be used by a (resolution) theorem prover. The theorems whose proofs are attempted provide PROLOG's input/output. Call a formula to be proved an "I/O formula." An I/O formula must be an existentially quantified conjunction. If it has no variables, the output is the TRUE/FALSE response of the theorem prover. Similarly, if an I/O formula with variables is not a theorem, the output is FALSE. Otherwise, an I/O formula is a theorem because particular constants do satisfy its unquantified form, and the PROLOG output in that case is the value of the satisfying constants. The input may be thought of as parts of the I/O formula not containing variables.

For example, the PROLOG program:

```
plus(X, [], X).

plus(X, b.Y, b.Z) <-- plus(X, Y, Z).
```

defines a predicate intended to hold of lists of b's if the third parameter is the unary sum of the first two, zero being represented by the empty list []. With the I/O formula

```
plus(b.b.[], b.[], Q).
```

the output is a value for Q (b.b.b.[]), and the input-output correspondence is straightforward. Because the addition function is invertible, any of the parameters can be calculated from the other two in the same way. With the I/O formula

```
plus([], b.[], b.b.[]).
```

the result is FALSE from the theorem prover, and similarly for

```
plus(X, b.b.[], b.[]).
```

The I/O formula

```
plus(U, V, b.b.[]).
```

yields three possibilities for (U, V) pairs:

```
b.b.[], []
b.[], b.[]
[], b.b.[]
```

but

```
plus(b.[], U, V).
```

has an unlimited number of satisfying pairs:

```
[], b.[]
b.[], b.b.[]
...
```

Because of the way the program is written and the resolution proof technique, the results from

```
plus(U, b.b.[], V).
```

are a kind of symbolic execution instead of an unlimited table:

```
V is b.b.U
```

and

```
plus(U, V, W).
```

results in triples (U, V, W) which are a mixture of generated test data and symbolic execution:

```
U, [], U
U, b.[], b.U
U, b.b.[], b.b.U
...
```

A programming style for PROLOG comes from logic. (The sample above is from the usual formulation of number theory, for example.) This style is difficult to characterize, but one variant recalls the Gist idea of generating more than the behaviors of interest, then pruning them. To define predicate P one instead defines Q that is less restrictive than P , then defines R to include further restrictions, and P as Q /\ R . For example, in the definition of the Kleene T-predicate for Turing machines, the idea of sequence is defined, and then successively restricted to sequences of instantaneous descriptions, then to such sequences that can legally arise from a machine, then finally to such sequences that begin and end properly [6]. The virtue is this kind of programming is that it decomposes a problem into easily understood parts. If PROLOG is used in this way, the addition of each predicate further restricting the solutions may be thought of as successive approximations to the problem specification, and the intermediate programs as partial specifications.

The only difficulty in testing PROLOG programs arises from inefficiency of the theorem prover: if there are several output values, some more difficult to discover, then the human user may become impatient, and judge the specification on only part of its behavior. However, this effect is not perniciously connected with unfinished specifications: it is unlikely that omitting restrictions will make easily found outputs the ones a user hopes for. Resolution is an unnatural computing device, in the sense that human beings find it difficult to predict how unification will proceed. A comparison with the concurrent-process languages is apt: the semantics of

PROLOG defines the output from an I/O formula to be all constants satisfying its unquantified form, and we fault the theorem prover for generating them too slowly. The semantics of concurrent computation defines correct output as any of the possibilities that arise from process activation sequences; if all possibilities were required, the same kind of exponential times would result. This is seen in Gist where the constraints may require many of the possibilities to be investigated. Indeed, a PROLOG interpreter can be thought of as a Gist specification in which all terms of the Herbrand universe are generated, with the PROLOG program as constraints. There seems as little reason for accepting any process sequence, when we do not understand all the possibilities, as for accepting any output from a PROLOG program, when we do not understand the resolution computation.

4. Discussion: Testing System Specifications

Testing always has a two-fold purpose. Initially, it is to find mistakes in a system, and aid in their correction. When no more mistakes can be found, the test data assumes the new role of establishing confidence that there are no more mistakes to find. In so-called "black box" (or "functional") testing the software's overt, external behavior alone is of interest, without regard for its internal structure. The difficulty with black-box testing in concurrent systems is that the required behavior is distributed in space and time. A system for process control, for example, is required to read sensors and drive actuators in a complex time sequence, so that generating realistic test cases may be difficult. The alternative to black-box testing is called "structural" (or "program based") testing, in which an attempt is made to investigate code usage. The idea is that mistakes must lie textually within the software sources, and will not be found unless all parts are exercised in some way. Structural methods differ in what they consider as "exercised;" for concurrent systems it must include trying different orderings for the processes involved.

For specification languages there are additional testing problems introduced by implementation and by unfinished specifications. A specification is an object in its own right, and its internal properties (such as consistency) are of interest. But eventually it will be implemented in an efficient way, perhaps with a software structure rather unlike the specification. Direct tests of the specification can be misleadingly successful in that an implementation that meets the specification can fail that same test. When the specification includes a portion that models the environment, and which is not implemented, the implementation may have to preserve the same structure as the specification in order that specification tests can be used. Testing an unfinished specification exacerbates these difficulties, because missing portions rely on human simulation of the system's behavior, which may be based on wishful thinking.

4.1 Structural Tests of Concurrent Software

An advantage of structural over black-box testing is that the generation of test data is methodical. For example, if the structural exercise criterion is that every statement must be executed, a programmer usually has little difficulty devising the needed data. The structural criterion that there should be some kind of coverage of the many possible patterns of process orderings is more difficult to achieve. For a given input, there are several common possibilities: (1) the software is supposed to synchronize its processes so that the process ordering is fixed; (2) although many process orderings are possible, the overt behavior of the system is not supposed to depend on which one occurs; (3) different behavior results from different

ordering. A test that attempts to vary the process ordering will have different outcomes for these cases; neither specification nor programming languages have ways to indicate the intent.

A kind of "stress testing" can be applied to concurrent software by tampering with processes' scheduling. Each process can be given the highest (or lowest) priority in turn; if the number of open scheduling decisions is small, all possible priority patterns can be tried, giving an exhaustive test of orderings. It should also be useful to try disciplines unlikely in practice, such as random scheduling, or scheduling by longest time since previous activation. In controlling the testing of concurrent software to meet structural coverage criteria, it may be necessary to use sophisticated tools [7].

Specifications probably should not include explicit information on scheduling (the Gist constraints and PAISLey timing information are an attempt to describe the "what" of scheduling decisions without the "how"). Implementations, however, may have to resort to fine control of scheduling to perform as required. This means that an implementation test may have very little ordering freedom compared to a specification test, and it may prove possible to cover the possibilities much better in the former.

4.2 Problems of Implementation

A specification expressed in a formal language is useful in communicating development decisions to users in the form of testable prototypes. However, the point is lost if successful specification tests (which convince a user that requirements are being met) become test failures for the final implementation. A more subtle difficulty is that the specification may describe both the environment and the computer system, while only the latter is implemented. If there are improper connections between the two parts of the specification, it may work where no implementation can. For example, it is clearly improper to have an environment process respond to the software state as in a "requirement" that a person not push some button when a variable has certain values. To include such interactions makes it impossible to test for their violation in the specification; the implementation is likely to have to deal with them early on.

A correct implementation of a Gist or PAISLey specification must intuitively agree with the specified behavior. The difficulty in precisely defining this idea is that "input-output" is woven into specifications by including processes that are part of the environment and thus not to be implemented. In Gist an environment demon alters the universe (for example by creating an object), a kind of "input" to which the remainder of the system responds, resulting in a changed universe that might be called the "output." In PAISLey, an environment process can match exchange functions with those in the system portion, the former's parameters then acting as "input" and the latter's as "output." In both cases the situation is complicated by timing: the demon or process can supply its inputs in a complex pattern. This kind of behavior is essential in the specification of embedded systems, whose environments do act in a patterned fashion.

An implementation will be judged by how well it responds to the real-world environment. Users should be making a judgement of how well the world is simulated in the specification, independent of the system part of the specification. Insisting on an independent environment prototype also insures that the interface to the system is proper--that the environment does not use or supply information it is not meant to. Neither PAISLey nor Gist controls this interface: the power of these languages hides it. For example, a Gist

environment demon could be constrained by a relation using system information; in testing the whole specification it would be very difficult for a user to detect that the proper behavior was the result of the world being careful not to strain the computer. But if the environment were tested separately, the improper relation would come up undefined, and the user has a chance of noticing it.

If there is an identified environment portion to each specification, one capable of output-input behavior in isolation, an implementation can be defined as correct if it interacts with the specification environment as the specification system does. That is, for each specification environment behavior, the implementation responses interleave with those of the environment to form at least one pattern that the specification system itself forms. This definition allows investigation of the relationship between successful tests of the specification and those same tests applied to an implementation.

When an implementation test is conducted, the specification should serve as an oracle--it should be able to answer unambiguously that the test result is or is not correct. This is possible only if all th~ process orderings of the specification are tried. (The nondeterminism of a concurrent implementation itself poses a distinct problem: all of its behavior sequences for each test input must be examined as well.) When an implementation is correct, it may be difficult to find the execution of the specification that shows it to be so. If it is incorrect, only exhaustive testing of the specification suffices to learn this. From another viewpoint, if a user has experimented with the specification and believes it to fulfill his requirements, if he has missed any of the behavior possibilities, a correct implementation can deliver a nasty shock.

Unfinished specifications only exacerbate all of these difficulties. It makes little sense to work with a partial environment specification, since the system is then responding to a world that will never be, and it is immaterial whether it can do so. When a partial system specification is being tested, there are more possible behaviors than finally intended, and this multiplies the chance of missing important ones. When the human user controls sequencing or supplies information that does so, it is likely that he will favor those sequences that were intended at the expense of unexpected ones, again increasing the chance that an implementation will be correct, but unacceptable.

Acknowledgement

This paper was written for a panel session at the Hawaii International Conference on Computer and System Sciences. Pamela Zave, Bob Balzer, and Jack Wileden are responsible for the work referenced, but of course not for my attempts to describe it. Without Bill Howden, this paper would not have been written, because he first said it would appear in the conference proceedings (sight unseen), and when it was ready changed his mind (sight unseen).

References

1. P. Zave, An operational approach to requirements specification for embedded systems, IEEE Transactions on Software Engineering SE-8 (May, 1982), 250-269.

2. P. Zave, Testing incomplete specifications of distributed systems, Proc. ACM Symposium on Principles of Distributed Computing, Ottawa, 1982, 42-48.

3. R. Balzer, N. Goldman, and D. Wile, Operational specification as the basis for rapid prototyping, ACM SIGSOFT Workshop on Rapid Prototyping, Columbia, Md., April, 1982.

4. R. Balzer, Design specification validation, Rome Air Development Center Technical Report RADC-TR-81-102, 1981.

5. R. A. Kowalski, Algorithm = logic + control, CACM 22 (July, 1979), 424-436.

6. R. G. Hamlet, Introduction to Computation Theory, Intext, 1974.

7. P. C. Bates and J. C. Wileden, EDL: a basis for distributed system debugging tools, Proc. 15th Hawaii International Conference on System Sciences, Honolulu, 1982, 86-93.