N00014–93–C–0213
FOURTH QUARTERLY REPORT

FORMAL SYSTEMS DESIGN & DEVELOPMENT, INC.
P.O. BOX 3004
AUBURN, AL 36831–3004
(205) 887 9444

N00014–93–C–0213

FOURTH QUARTERLY REPORT

Table of Contents:

FormalSystems

N00014–93–C–0213
Fourth Quarterly Progress Report

Michael Goldsmith
Formal Systems Design & Development, Inc.

December 16, 1994

Summary
This Document summarizes the progress to date in the Office of Naval Research SBIR Project N00014–93–C–0213 *Embedded Transputer-based System Design* and indicates the expected direction of the Research and Development in the following periods.

1 Overview

The level of effort expended is currently broadly on track both at Formal Systems and at the Charles Stark Draper Laboratory (Draper) and Formal Systems (Europe) Ltd. This quarter's work has spilled somewhat into the following period, due to staff resource problems arising from circumstances outside our control. Additional effort within the project fifth quarter is expected to bring progress back into line with the plan in a reasonable time.
With regard to the personnel changes foreseen in the previous report:

- Neil Brock, the technical point-of-contact at Draper, left as anticipated. Ms Donald, however, also left Draper over the summer. The new technical point-of-contact is Richard Harper [(617) 258-2243], the Senior Engineer leading the technical development of Draper's current fault-tolerant computer systems.
- Dr Richard Chapman (of Auburn University) is continuing to contribute to the project on a part-time basis.

The main areas of activity and achievement during this period are:

- Completion of the first iteration of work on a prioritized model for CSP suitable for justifying discrete analysis of real-time behavior.
- Prototype implementation of an extension to **FDR 2** to support this model.
- Design, analysis and modeling of the communication and voting architecture of a Transputer Fault-Tolerant Processor node with particular provision for the interaction with its task scheduling.

These are discussed in more detail in the following sections and the accompanying Deliverables.
The current status of Deliverables is summarized in Table 1.

Deliverable		Due	Status
D2.1	Detailed natural-language problem statement	End Q1	Delivered Q2
D2.2	Formalization of single-lane scheduling problem and of fault tolerance requirements	End Q1	Delivered Q3[1]
D1.1	Initial requirements definition for real-time modeling extensions to FDR	End Q2	Delivered Q2
D2.3	Idealized (single-lane) scheduler model	End Q2	Delivered Q3[1]
D2.4	Fault models and redundant scheduler correctness criteria	End Q3	Delivered Q3
D1.2	Prototype software for discrete real-time extensions to FDR	End Q4	Delivered Q4
D2.5	Initial process-algebraic solution and Draper appraisal of scheduler models	End Q4	Part delivered Q4
D1.3	Prototype Software for Continuous Real-Time Extensions to FDR	End Q5	Deferred
D1.4	Appraisal and Revised Requirements for Discrete Real-Time Extensions to FDR	End Q5	On schedule
D1.5	Translation and Interface Tools Requirements Definition	End Q5	In progress
D2.6	Timing Requirements Analysis for Scheduler	End Q5	On schedule
D1.6	Appraisal and Revised Requirements for Continuous Real-Time Extensions to FDR	End Q6	Not yet started
D1.7	Prototype Software for Translation and Interface Tools	End Q6	Not yet started
D2.7	Initial Prototype Transputer/occam Implementation and Verification of Conformance	End Q6	Not yet started
D1.8	Revised Code and Full Draft Documentation/Justification of Tools	End Q7	Not yet started
D2.8	Revised Prototype Transputer/occam Implementation and Architectural Specification of Potential VLSI Realizations	End Q7	Not yet started
D1.9	Final Report on Theoretical and Software Tool Developments	End Q8	Not yet started
D2.9	Final Report and Appraisal of Fault-Tolerant Scheduler Demonstrator	End Q8	Not yet started

Table 1: Deliverable schedule

Note 1: As anticipated, the delayed deliverables D2.2 and D2.3 have been consolidated into a single document.

2 Theory and Software Tools

The major goal of this project is to establish a viable route from specifications in Hoare's *Communicating Sequential Processes* (CSP) [4] and its real-time variants [5, 2] to implementations of real-world, substantial real-time and/or fault-tolerant systems. The initial concentration of effort under this head has been directed towards closing the gap between the current real-time specification and hand-crafted verification available within Timed CSP, on the one hand, and the available highly efficient mechanized verification and development aid for untimed CSP systems which is presented by the Formal Systems (Europe) Ltd model checking tool, **FDR**, and the new generation **FDR 2**.

The approach taken addresses this problem from both ends: finding a model and style of specification which allows description of (necessarily discrete) real-time behavior in terms of events, which can be handled by **FDR**; and expanding the capabilities of **FDR** to deal with the greatly increased complexity (in terms of number of states) that appears when consideration of the passage of time is introduced to formerly atomic events, and to allow ready adaptation to non-standard variants of operational semantics and refinement. Both these issues have been addressed within the framework of the second-generation tool, **FDR 2**. Progress to date is reported in Deliverable D1.2, which accompanies this report.

The apparent tractability of the kind of problems arising from the Demonstrator Application under the discrete modeling of time is such that we propose to concentrate our attention on that approach for the present, deferring the study of continuous real-time tools until later in the project.

3 Demonstrator Application

The demonstrator application is to be a verified real-time fault-tolerant scheduler, for a machine such as the Draper Transputer Fault-Tolerant Processor (TFTP).

Significant features of recent developments include:

- Clarification of arguments based on symmetry which can be used to establish properties of the full TFTP system from properties of a single voter. This work is sufficiently established that we feel a formal mathematical proof of the approach could be given. It is a result which will be particularly important in the future development of models which include more detail about the operating mechanisms of their components. It has already assisted in the rest of this work.

- We have presented models of the FTP consistency algorithm which include explicit timing information in both synchronous and semi-asynchronous models. These models include sufficient information about the communication mechanism to investigate the need for non-blocking and sacrificial buffers. We feel that these models approach the "Synchronous replicated" and "Asynchronous distributed" views of [1], although they still involve significant abstraction from the way in which the processing and voting elements operate, and the issue of establishing co-ordinated global timing has still to be addressed in detail.

- Moving toward a less abstract model bearing a closer resemblance to the implementation, we have gained significant understanding of the problems faced in several key areas. These include:
    - tolerance of transient faults,
    - recovery after transient errors, and
    - the benefits to be gained from temporal redundancy and permuted scheduling.

  The modeling which we have completed in this area is still highly abstract, but it provides important framework elements, and highlights those areas which place additional emphasis on new theories and tools.

The resulting document is being submitted to Draper for comment, and the models will be revised as necessary in the light of feedback.

References

[1] N.A. Brock. Real-Time Scheduler: Natural Language Problem Statement. Technical report, Charles Stark Draper Laboratory, Inc., 1994. Deliverable D2.1 of SBIR N00014-93-C-0213, in [3].

[2] J. Davies. Specification and Proof in Real-Time Systems. Programming Research Group Technical Monograph PRG–93, Oxford University Computing Laboratory, Oxford, England, 1991.

[3] M.H. Goldsmith et al. *N00014-93-C-0213: Second Quarterly Report.* Technical report, Formal Systems Design and Development, Inc., P.O. Box 3004, Auburn, AL 36831-3004, 1994.

[4] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[5] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. In *Proceedings of ICALP'86, LNCS 226.* Springer-Verlag, 1986. Also appears in Theoretical Computer Science 58 (1988) 249-261.

This is Release 1.0 of this document, last modified by Michael Goldsmith at 17:50:08 GMT on December 16, 1994.

FormalSystems

Inside **FDR 2**

P.H.B. Gardiner M.H. Goldsmith

Formal Systems (Europe) Ltd
Formal Systems Design & Development, Inc.

December 16, 1994

Summary

This document introduces the internals of the second-generation of the **FDR** tool.

1 Failures-Divergence Refinement

Failures-Divergence Refinement is an algebraic property that may hold between two abstract processes. It is by checking whether this property holds between two state-machine descriptions – and helping to determine why not, in the case that it fails to hold – that the Formal Systems (Europe) Ltd tool **FDR** operates. Both CSP and its theories prove remarkably well-suited for this kind of mechanical model-checking analysis for a variety of reasons:

- the semantics, and equivalence, are based around the idea of refinement;
- the ideas of parallel composition and hiding are separate, so that multiple processes can synchronize on events and enforce constraints, and hiding can be used as abstraction;
- CSP includes a wide range of operators, both ones representing real modes of constructing processes and ones which, while pointless or difficult to implement, are useful in building specifications.

These have meant that it has been possible to build a fast refinement checker which can be used for the great majority of correctness proofs one is likely to want, and that the language is able to represent complex systems succinctly and clearly.

This is an inter-company report intended as an adjunct to Deliverable D1.2 in the Office of Naval Research SBIR Project N00014–93–C–0213 *Embedded Transputer-based System Design*. Much of Section 1 is derived from the fuller discussion of model-checking CSP in [7].

1.1 Models of CSP

The standard model for (untimed) CSP is the *failures/divergences* model, where each process is represented by two sets of observable behaviors:

- *failures* are pairs (s, X) where s is a finite sequence of actions which the process can perform (a *trace*) and X is a set of events it can refuse after s;
- *divergences* are finite traces after which the process can perform an infinite sequence of consecutive internal actions.

A more operational view of CSP (and other process languages) uses the notion of transition systems. The model is expressed as a *transition relation*, $\rightarrow$, which relates process states to their possible actions and the states which result from performing those actions. If a process P can perform an action a and subsequently behave like Q we write $P \xrightarrow{a} Q$. If the action is internal (and therefore independent of the process's environment) we write $P \xrightarrow{\tau} Q$, as in CCS and similar languages. Models of this form can be view as graphs whose nodes are processes and whose (labeled) arcs represent actions. Because such structures can be represented and manipulated efficiently, this representation is very attractive for mechanized analysis. The actual model used does not alter the results of any analysis, because a congruence exists between the two frameworks. Consequently we will tend to use the transition graph representation of processes for mechanical manipulation while the observational model still motivates and guides the underlying theory.

We will therefore concentrate on deciding questions about finite directed graphs where all edges are labeled with an action, either τ or visible. These systems are called *finite labeled transition systems.*

1.1.1 Refinement

We say that a process P with failures F and divergences D *refines* another, P' with observations F' and D', if and only if any observation possible for P is also possible for P'. Formally, we write

$$P' \sqsubseteq P \qquad \text{iff} \qquad F \subseteq F' \quad \text{and} \quad D \subseteq D'$$

This containment can be used in a number of ways:

- It implies that any trace of P must be a trace of P'. If we consider P' to be a specification which determines possible safe states of a system, then we can think of $P' \sqsubseteq P$ as saying that P is a safe implementation: no wrong events will be allowed.
- Having included the refusal information allows us to constrain the events which an implementation is permitted to block as well as those which it performs, and thus to capture liveness properties.
- Further, using divergence gives two major enhancements: we may analyse systems which have the potential to fail (including failure by livelock), and assert that failure does not occur in the situations being considered, and we may also use divergence in the specification to describe "don't care" situations.

Important facts that are exploited by **FDR** include the following:

1. The least process under the refinement order, $\perp$, equates to any process that can diverge immediately (i.e., without performing any visible actions first), and equivalently to the most undefined process.

2. The refinement order is complete and its maximal elements are the *deterministic* processes: divergence free and, after each trace s, only able to refuse those events that it cannot communicate after s.

3. Each standard CSP operator can be defined as an operator over sets of failures and divergences. Using the refinement order and least fixed points for the semantics of recursion gives a denotational semantics that is congruent to the operational semantics.

Another process of particular interest is that which can exhibit any trace and refusal combination, but does not diverge. This process, $CHAOS$, is refined by any non-divergent (i.e. livelock free) process.

In practice, the statement $Q \sqsubseteq P$ states that P can never be observed to behave in a way not allowed by Q. We thus tend to term Q the "specification" and P and "implementation". In other notions of refinement we might term Q an abstract and P a concrete model, and indeed CSP refinement will support a step-wise development style.

1.2 Mechanical Verification

We now consider how we might check a statement of refinement (like those above) mechanically. The expressive power of the CSP language is such that not all such problems are decidable, so we must restrict our range of problems to some practically useful set.

1.2.1 Tractable Problems

For simplicity, in the original version of **FDR** it was possible to deal only with processes whose operational semantics are *finite state*. It is theoretically feasible, by exploiting some form of lazy evaluation, to relax that restriction on the specification (left-hand) side; **FDR 2** is developing support for this[1]. This means that, as the operational semantics is unfolded, only finitely many process terms are generated. Thus,

$$P = a \rightarrow ((b \rightarrow STOP) \,\square\, (c \rightarrow P))$$

is finite state, but on the other hand,

$$Q = a \rightarrow (Q \,|||\, (b \rightarrow STOP))$$

is infinite-state, and fundamentally so since it is a process which can always communicate a, and will communicate b provided this will not make the number of b's so far exceed the

[1]This is desirable so that, for instance, a most-nondeterministic buffer, with no preset bound on its capacity, can be used to represent the property "*is a buffer*"; while it is usually quite simple to calculate a bound on the buffering of a system, it is an inelegant complication of the specification to be required to.

number of a's. There are some simple rules to help determine what classes of CSP process are likely to be finite state.

1.3 Checking strategy

Our approach to actually checking the refinement is divided into two parts: normalizing the specification, and then checking the implementation against the resulting normal form. The reasons for this division and the mechanisms used in each phase are outlined below.

1.3.1 Normalizing a transition system

The transition systems arising from CSP descriptions typically contain a high degree of nondeterminism, in the sense that after any trace s of visible actions there may be many states of a system which the process might be in. This can happen both because of the existence of invisible actions and because of the branching that occurs when a node has two identically-labeled actions, whether visible or invisible. Any method for deciding refinement between these systems will have to keep track of all the states reachable at the specification side on a given trace s, since it is merely necessary than every behavior of the implementation on s is possible for one of these. In addition, any method we devise will need a way of telling when a large enough set of traces have been tried to establish refinement.

Life would thus be easier if there were exactly one state corresponding to each possible trace. This can be achieved by transforming the original specification transition system to an equivalent *normal form*. The idea of a normal form for CSP processes has its origins in [2], where it is shown that each finite CSP term is equivalent to one in the following normal form:

- The least defined process, $\perp$ is a trivial normal form;
- all others take the form
$$\bigsqcap_{A\in\mathcal{A}} x : A \rightarrow P(x)$$
for $\mathcal{A}$ a convex set of sets of events such that $\bigcup\mathcal{A} \in \mathcal{A}$, and each $P(x)$ a normal form which depends only on x, not on A.[2]

As an example, we will normalize the a 3-place (data-abstracted) buffer process B_3 defined:

$$B_3 = COPY \gg COPY \gg COPY$$

where

$$COPY = in \rightarrow out \rightarrow COPY$$

(The operator '$\gg$' is a form of parallel composition that connects the *out* channel of the left-hand argument to the *in* channel of the other, and hides the result.) The transition system of each this process, and the corresponding normal form, is shown in Figure 1. B_3

[2]This means that $\mathcal{A}$ can be represented effectively by its minimal elements (the *minimal acceptances* of the process, which are the Σ-complements of its maximal refusals), together with the maximum element (its *initials*).

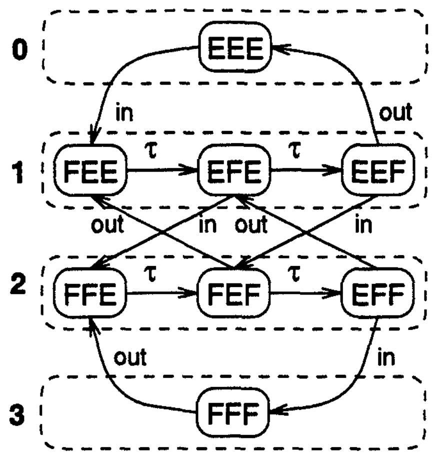

Figure 1: A simple normal form

has 8 states, corresponding to each of the component processes being in state $COPY = E$ or $out \rightarrow COPY = F$. Its normal form has 4 states, one of each possible number of "items" this "buffer" is holding (**0**,**1**,**2** or **3**).

In actual fact, collecting together states reachable after a given trace in this way, while sufficient to allow our checking algorithm to proceed, does not produce a strict *normal* form. In order to achieve this, we need (and FDR employs) a straight-forward compression algorithm to merge states from the *pre-normal* form which actually share the same future behavior. (In the case of the buffer example, the pre-normal and normal forms are identical.)

The complexity of normalization Given that the pre-normalization process builds a transition system over the sets of nodes from the original system, there is the possibility that the normal form will be exponential in the size of the original system. This can indeed occur, as can be shown by a number of simple examples. Fortunately there are two mitigating factors that work in our favor, making this particular obstacle more-or-less disappear.

1. "Real" process definitions simply do not behave as badly as the pathological examples. In practice it is rare for the normal form of a naturally-occurring process to have more states than the original transition system. Indeed, the normal form is frequently significantly *smaller*, offering scope for intermediate compression.

2. It is only the *specification* end of the refinement that we have to normalize. In practice the simpler process is usually that one rather than the implementation. Frequently, indeed, the specification is a representation of an abstract property such as two events alternating or deadlock-freedom, and has a trivial number of states. One would usually expect that a process playing the role of a "specification" is reasonably clearly and cleanly constructed, with understandable behavior. These aims are more-or-less

inconsistent with the sort of nondeterminism that leads to an explosion in the normal form.

1.3.2 Checking refinement

Once the specification end of a refinement check has been normalized, the following two phases are necessary to establish failures/divergence refinement.

1. Establish which states of the implementation transition system are divergent, marking them as such.
2. Model-check the implementation, thus marked, against the normal form.

A state P is divergent if, and only if, the directed graph formed by considering only τ actions has a cycle reachable from P. There are two standard algorithmic approaches to this problem: computing the transitive closure of the graph or taking advantage of the properties of depth-first search (DFS). The **FDR** tool uses variations on the latter approach.

In the model checking phase we have to discover whether all the behaviors of each implementation state are allowable in all the normal form states such that they have a trace in common. This is done by exploring the cartesian product of the normal form and implementation state machines. Each state in the product is expressed as a pair $\langle \nu, w \rangle$ where ν is a state in the normal form and w is a state in the implementation.

Conceptually, we maintain a set of checked pairs and a sequence (pre-ordered by the length of trace) of *pending* pairs; initially the former is empty and the latter is the singleton of the pair of initial states of the two systems. Until it is empty, we repeatedly inspect pairs from *pending*. The pair $\langle \nu, w \rangle$ checks if

1. the normal-form state ν is divergent; *or*
2. the implementation state w is non-divergent, *and*
   (a) the set of initial actions of w is a subset of those of ν, *and*
   (b) *either* w is unstable (i.e., it has a τ-action), *or* the set of actions it refuses (the complement of its initials) is a subset of one of the maximal refusals of ν.

If the pair does meet these conditions, we add it to *checked* and proceed as follows: the set of all pairs reachable from $\langle \nu, w \rangle$ and not in *checked* is added to *pending*. A pair $\langle \nu', w' \rangle$ is *reachable* from $\langle \nu, w \rangle$ if either

1. $w \xrightarrow{\tau} w'$ and $\nu = \nu'$; *or*
2. $w \xrightarrow{a} w'$ and $\nu \xrightarrow{a} \nu'$ for $a \neq \tau$.

Note that whenever $w \xrightarrow{a} w'$, then this ν' must exist and be unique.

If a pair is found which fails to check, then the proposed refinement does not hold. If the above process completes without finding such a pair, then refinement does hold. In order to simplify debugging, we are now in a position to report the shortest possible sequence of actions that can lead to such an error, when one is found. This is achieved by the breadth-first search (BFS) implied by the ordering on *pending*.

If no error is found, one can justify the claim that refinement always holds either operationally or abstractly. Operationally, if refinement fails, then it must hold because the implementation has some behavior that is banned by the normal form. There is therefore some sequence of actions which exhibit this, potentially bringing the implementation into some state w where it can behave illegally. It is clear that the unique normal form state ν corresponding to this trace is such that $\langle \nu, w \rangle$ will be found in the search above, leading to a failure to check. Abstractly, one can show that the refinment checking process is simultaneously formulating and proving a sort of mutual recursion induction over the state-space of the implementation.

1.4 Implementation Issues

1.4.1 Machine-Readable CSP

CSP has been, as a "blackboard" language, a rather variable notation, with some parts being used differently in different schools and some parts being rather under-defined. Since it is written, essentially, as a series of algebraic expressions using a variety of peculiar-looking operators, it does not look like the sort of programming language one usually types into a computer.

FDR emerged as part of a general effort to make tools available for CSP. There was thus an obvious need for a standardized syntax, parser and type theory/checker for the language. The work[3] to develop these has been led by J.B. Scattergood [8], and has gone on hand-in-hand with that on **FDR** which, together with several other tools, uses them.

The objective of this work has been to preserve as much as possible of the form, spirit and flexibility of the blackboard language, while adding sufficient structure to allow for standardized mechanical interpretation. For portability, it was decided to do all of this within the confines of an ASCII syntax – where the chief compromise is in the representations of the operators – though this does not preclude the building of more elaborate display and printing facilities on top of it.

Most of the decisions in the design of the syntax revolved around the sub-process objects such as events, their atomic components and parameters to processes (which can be thought of as process state). Unless we change the CSP semantics of termination and sequencing, the treatment of these objects is necessarily declarative: there is no assignment or similar construct that can change the value of an existing variable within its scope. It also affects the style of CSP programs, which become *scripts* of definitions (of a mixture of processes and other objects) in the style of functional programming languages such as Haskell and that of [1].

More specific features of the machine-readable syntax include:

[3]The CSP parser and typechecker were produced under an ONR-sponsored project at Oxford University, and are freely available to anyone interested in producing tools related to CSP.

1. In [4], every process has an intrinsic alphabet, with these being important for the semantics of the parallel operator. A pair of processes must synchronize on events in the intersection of their alphabets. We have found it more convenient to supply either these alphabets directly when using the parallel operator or, more usually, to define the interface set of each parallel composition: $P \underset{A}{\|} Q$, written `P [|A|] Q` in ASCII, makes P and Q synchronize on elements of A.

2. Each event is made up from a constructor representing its name and a number of "data" components from a fixed list of types (which may be empty). Each "channel" thus has a possibly trivial cartesian product type. The events are formed using infix dots, sometimes by `?` representing input and binding the identifiers to their right until the occurrence of a `!` representing output. Thus the event `a.1?x!y` is a communication over channel `a`, whose first and last components are fixed and the middle one is open and will bind `x` to the value input. The original **FDR** demands that all channels used are specifically declared through a pragma mechanism; as the parser acquires a type system, this will also be supported in **FDR 2**.

3. The syntax has specific support for booleans, numbers, sequences, tuples and sets. Any undefined identifiers (ones not bound to anything specific) are treated as tokens or constructors introduced by the user.

2 Structure of the FDR 2 core

The second-generation implementation, **FDR 2**, differs in several significant respects from the original. These are mainly aimed at increasing the flexibility and scope of the tool.

- The language of implementation for (at least) the computation and memory intensive tasks – previously carried out by **C** programs – is now **C++**.
  - Inheritance and virtual methods allow multiple representations of processes to co-exist and interact without necessitating multiplication of the coding of the high-level algorithms involved.
  - Abstract interfaces permit link-time extensions to the repertory of representations without necessitating modifications to the existing source, recompilation or even access to anything other than interface header files.

- The two state-machine descriptions supported by **FDR** were an explicit representation of a (pre-) normal form (the output from `normal` and the first input to the various `refine` programs) and moderately compact encoding of (potentially parallel-) machine transitions between binary state vectors, with "don't-care" (and "leave unchanged") masking of components not taking part in the synchronization.

  These are both supported (interchangeably) in **FDR 2**, but in addition two families of representation address the problem (discussed in [5]) of the multiplied numbers of transition rules arising where two or more components synchronize on an event which they can each perform in a number of states.

- Each syntactic operator (especially, but not necessarily only, those which **FDR** regards as "high-level") can be regarded as defining a new representation, where the arguments (both process and non-process) are recorded as data members of a new derived class, and where the methods yielding the operational semantics of the constructed process infer its behavior from those of its components.
- As a generalization of this, a "supercombinator" can be compiled from a tree of such operators, encoding all the information necessary to infer the whole operational behavior from the ability of the ultimate components to engage (or refuse to engage) in particular events in their current state.

In each case, the size of the representation is reduced to approximately the sum (rather than, in the worst case for a state-mask machine, the product) of the sizes of its "leaf" machines. The time to calculate the results of the semantic methods is similarly reduced.

- While the failures-divergences model is the canonical semantics for CSP, in many circumstances the distinctions it draws are finer than is needed to capture the intended property. Safety properties, in particular, depend only upon the possible traces of the specification and implementation, and discount the information that failures give about the presence or absence of nondeterminism. Similarly, in cases where other reasoning gives grounds for believing some sort of fairness condition holds, it is correct to ignore stabilizable divergence [6].

  Rather than coding separate, but very similar, routines to carry out normalization and refinement-checking for each of the models in the hierarchy, it is equivalent (and not noticeably less efficient) to implement the retract mapping which projects a process in the failures-divergences model onto the more abstract model and brings it back through a canonical embedding. Applying such model-changing transformations to both sides[4] of a refinement query yields the answer to the more abstract question.

- A wide range of compression techniques can be applied to labeled transition systems: factorization by weak or strong bisimulation equivalence, and collapsing τ-loops, chains or diamonds, for example. Applying these to all or part of the specification, or to carefully selected components of the implementation, promises to yield very significant reductions in the complexity of deciding many refinement questions (this is one of the areas of research currently under active investigation).

- While the more structured representations of processes undoubtedly are more compact, they necessarily carry a degree of computational overhead relative to a fully tabulated ($O(1)$ access) machine. Thus where space allows – which almost certainly includes most genuine "leaf" processes in a highly structured machine – it is desirable

[4]It might be adequate semantically to apply, say, the retract which maps any process to the most nondeterministic one with the same trace set only to the specification side; but to do so to the implementation as well simplifies the process which needs to be checked.

to evaluate the transition system once and for all. This too fits comfortably into the **FDR 2** framework.

3 Fundamental types

In order to be able to produce customized versions of **FDR 2**, a certain grasp is required of the classes and methods forming the internals of the C++ implementation. Of these, undoubtedly the most important is the class `ISM` (Indexed State Machine), but some other ancillary classes are required.

3.1 Type Event

Currently, the only supported type of `Event` (representing the labels, visible or invisible, on the transition relation) is a simple integer. The header file `event.hpp` declares

```
typedef int Event;

int is_tau(Event);

Event mk_tau(Event);
```

These functions satisfy `is_tau (mk_tau (e)) == 1` for every `e`. For backwards compatibility with **FDR**, `is_tau (999) == 1` is also guaranteed, but apart from these cases `is_tau (e) == 0`.

3.2 Class Node

Class `Node` is essentially an opaque class with equality and ordering predicates. Its primary use is to denote a particular state within a state-machine representation, as will become clear in Section 3.4.

It also supports methods for storing any data it contains efficiently in memory and later retrieving it; this is important as sub-classes allow tuples and sums of nodes to be embedded in the same space (to reflect the state of compound machines in the state of their components), as well as scalar values and the bit-vector representation used for **FDR**-style state-mask machines.

Monadic `ISM` to `ISM` constructors can safely use the `Node`s of their argument process without worrying about any internal structure. An appropriate blend of reference counting and deep copying is used to maintain a consistent relationship between construction, assignment and deletion of `Node` objects.

3.3 Classes Set<T> and Iter<T>

A templated family of `Set` classes is provided: among instances of particular importance are `Set<Node>`, `Set<Event>` and `Set<Set<Event>>`, as we will see in Section 3.4. `Set`s too are arranged to behave sensibly with respect to assignment and modification.

The public interface allows the standard operations:

```
template<class T>
class Set {
  friend class Iter<T>;
  ...
public:
  int empty() const;
  int cardinality() const;
  int member(const T&) const;

  void add_singleton(const T&);
  void add(const Set<T>&);

  friend Set<T> setunion(const Set<T>&, const Set<T>&);
  friend int subset(const Set<T>&, const Set<T>&);
  friend int proper_subset(const Set<T>&, const Set<T>&);
  friend Set<T> singleton(const T&);
  friend Set<T> setinter(const Set<T>&, const Set<T>&);
  friend Set<T> setminus(const Set<T>&, const Set<T>&);
} ;
```

as well as con- and de- structors, assignment, comparison and output operators.

The approved method for extracting the members of a set is to use an *iterator* of class Iter<T>. This makes efficient access to the internal representation of the Set object available through a simple interface:

```
template<class T>
class Iter
{
  ...
public:
  const Iter& operator=(const Set<T>&);
  int going() const;
  void next();
  const T& here() const;
};
```

The use of this class is illustrated by the following invariant:

```
  Set<T> s = ..., t;
  for (Iter<T> i=s; i.going(); i.next())
    t.add_singleton (i.here());
```

establishes s == t.

It may be noted that a `Set<T>` is effectively sorted with respect to (a required overloading of) the function `int compare(const T&, const T&)` and the corresponding `Iter<T>` releases them in ascending order without duplicates across calls of `next()` (so that

```
Iter<T> i = s;
T a = i.here();
i.next();
T b = i.here();
```

establishes `compare(a, b) < 0`).

3.4 Class ISM

The `ISM` class provides an abstract simulation interface for the operational semantics of a state machine. There are four fundamental questions which can be asked of a CSP process:

1. Is it divergent?
2. What events can it engage in?
3. What degree of nondeterministic choice is it allowed among these events?
4. What range of processes can it evolve into after engaging in some particular event?

This last question might naively be answered by returning a set of processes, each represented as a rooted transition system perhaps; and in general this may be the best which can be achieved. The algorithms set out above, however, rely on being able to determine whether a particular state has been seen before; and in the finite-state universe (and the relaxations of finiteness that we will allow) a more efficient encoding is possible. The property of its representation which distinguishes an `ISM` from a more general state-machine is that the object represents an entire machine, with the behavior of each state made accessible through a (hopefully relatively small) name of class `Node`.
The class `ISM` itself is abstract: it provides an interface of virtual functions to which particular incarnations must supply definitions, together with a number of derived functions to which more efficient implementations may be supplied if the derived representation will support them.

As a data-less class, it only makes sense to refer to pointers to an `ISM`, rather than to objects of the base class. In order to avoid scope/extent errors, raw pointers are eschewed in favor of a smart, reference-counting, class `ISMRef`. This is an instantiation of a templated `Pointer<T>` (technology which underlies the other reference-counted objects already described), which requires that its template argument be derived from a simple class `Counter` (which manages the reference count, destructing the object when it reaches zero). The `operator->` is overloaded to make these smart pointers otherwise indistinguishable from the built-in type.

```
class ISM : public Counter
{
  friend class ISMRef;
  virtual NSM* viewNSM() const;
public:

// Mandatory methods
  virtual Set<Event>         alphabet() const = 0;
  virtual Node               root() const = 0;
  virtual Set<Event>         initials(const Node&) const = 0;
  virtual Set< Set<Event> >  minAcceptances(const Node&) const = 0;
  virtual int                divergent(const Node&) const = 0;
  virtual int                markedDivergent(const Node&) const = 0;
  virtual Set<Node>          afters(const Node&, const Event&) const = 0;

// Derived methods
  virtual Set<Node>          states() const;
  virtual Set<Node>          tauClosure(const Set<Node>&) const;
  virtual Set<Event>         powerInitials(const Set<Node>&) const;
                             // tau not in powerInitials(s)
  virtual Set< Set<Event> >  powerMinAcceptances(const Set<Node>&) const;
  virtual int                powerDivergent(const Set<Node>&) const;
  virtual Set<Node>          powerAfters(const Set<Node>&, const Event&) const;

// Allowed not to define an output routine, but default is then error
  virtual void               write(ModeOut&) const;
};
```

The intended interpretation of the mandatory methods is as follows:

- `alphabet()`

  A set of events which includes all the non-τ labels on the transition system being encoded. It is useful only to provide an approximation to Σ, the domain of discourse, if debugging messages are desired in terms of refusals, rather than acceptances.

- `root()`

  A process is fully represented by an `ISM` (the transition system) together with a distinguished `Node` marking which state within it is "current". The initial starting point is encoded within the machine and accessed through this method.

- `initials(n)`

  The set of labels on any arc from node `n` in the transition graph.

  In encodings which have not eliminated τ-transitions, τs (that is, events `e` for which `is_tau(e) == 1`) will occur in this set if they label such arcs.

- `minAcceptances(n)`

  A set of pairwise-incomparable subsets of `initials(n)`, representing combinations of visible events which cannot be refused by node `n`.

  In a pure transition system, this will always be either empty (if the node has a τ transition) or the singleton set {`initials(n)`} (if it does not); in a normal-form representation of a non-divergent state it will be the minimal elements of the $\mathcal{A}$ of Section 1.3.1. Other representations may have both τ-transitions and non-empty minimal acceptances for the same node: this represents that other τ-transitions to stable states in the pure system have been collapsed onto the node at this level of representation.

  No τ occurs in any element of the set.

- **`divergent(n)`**

  Non-zero if the state represented by node `n` is divergent (that is, is reachable by a trace which is a divergence in the observational semantics).

- `markedDivergent(n)`

  Non-zero only if **`divergent(n)`** non-zero, and whenever this holds and there is neither a cycle of τ-arcs nor another **`markedDivergent`** node τ-reachable from `n`.

  This is not strictly part of the necessary simulation interface, but is a concession to efficiency. Some implementations may have recorded annotations on the divergence of nodes; others will need to explore τ-connected components of the transition graph looking for cycles. In the case of a machine built by hiding events of an underlying machine, for instance, both these may contribute: the underlying machine may have divergences not deducible from its transition graph (it may be normalized, for example), while the hiding may introduce new τ-loops. But if determining whether a node of the underlying machine is divergent itself involves τ exploration, inquiring of each node might result in a quadratic number of such inquiries, if only the **`divergent`** method were available. This method allows the two questions to be separated efficiently.

  Some implementations may also cache the deduced divergence of their nodes as they are calculated; they are free to return that information through this method.

- **`afters(n, e)`**

  The set of all nodes which are at the end of an arc from `n` labeled by `e` in the transition graph, including τ arcs if appropriate to the level of the implementation. For any reachable node `n`, and any `e` in **`alphabet()`**, **`afters(n, e)`** is non-empty precisely if `e` is a member of **`initials(n)`**.

  If `n` has not arisen directly or indirectly from calls to the **`root`** and **`afters`** methods of the machine in question, the results of this (and all the other methods taking a `Node` argument) are undefined. Similarly, no guarantees are made if `e` lies outside the universe of discourse.

The derived methods are values which most machine types will calculate in the same way from their instances of the mandatory methods, but which some may be able to evaluate through efficient short-cuts. The default definitions supplied with this base class, however, are available and often adequate.

- `states()`

  The set of all nodes reachable by a finite series of calls to `afters` from the `root` node; calculated by exploration. (Machines which use an initial segment of the natural numbers for their nodes and which know the high-water mark can obviously finesse this!)

- `tauClosure(s)`

  The set of all nodes reachable by a finite sequence of τ-arcs from any node in `s`. (For machines which have eliminated τ-transitions, this is simply `s`.)

- `powerInitials(s)`

  The union of `initials(n)` for each node `n` in `s`, with τ removed (since this is designed for application to the results of `tauClosure`).

- `powerMinAcceptances(s)`

  The minimal elements of the union of `minAcceptances(n)` for each node `n` in `s`.

- `powerDivergent(s)`

  Non-zero if `divergent(n)` is non-zero for any `n` in `s`.

- `powerAfters(s, e)`

  The union of `afters(n, e)` for each node `n` in `s`.

The private `viewNSM` method is discussed in the next section. Discussion of the `write` method is deferred to Section 3.6.

3.5 Class NSM

```
class NSM : public ISM
{
  friend class NSMRef;
  NSM<Event, Node>* viewNSM() const;
public:
  // ISM members defined in NSM
  virtual Set<Node>   afters(const Node&, const Event&) const;

  // Version of 'afters' giving unique result.
  virtual Node after(const Node&, const Event&) const = 0;
};
```

Another abstract class derived from `ISM` is `NSM` (Normalized State Machine) which gives the interface to normal-form machines.

There is essentially only one difference between them: it is known *a priori* that each event labels at most one arc from any given node, so it is reasonable to implement a method **`after(n, e)`** yielding the unique node reached from `n` by an `e`-transition (for any `e` in **`initials(n)`**).

There is no reason why a normalized machine cannot be viewed as an `ISM`, and the normal C++ conversions allow the casting of an `NSM*` up to an `ISM*` (or in our case, from a "smart" `NSMRef` pointer to an `ISMRef`). To go the other way, however, is not so straightforward; it is certainly desirable to be able to recognize a normalized machine, rather than have to re-normalize it, but it is not safe to allow arbitrary machines to be viewed as `NSM`s. The `viewNSM` method provides a technique for allowing this "downcast" precisely when it is safe: the default definition in `ISM` is to return a null `NSM*`, whereas the specialized definition in `NSM` is as the identity function. This whole is encapsulated in two **`friend`** functions to `ISMRef`,

```
NSMRef viewNSM(const ISMRef&);
int null(const ISMRef&);
```

so that `null(viewNSM(ism))` is zero just when `ism` is an object of a class derived from `NSM`.

3.6 Input/output mechanisms

FDR 2 incorporates a library of routines providing access to structured files, in either text or binary representations. These include methods for reading and writing integers and pairs of integers, sets of the same, bit-vectors and tagged lists.

```
class ModelIn : private ifstream
{
  ...
public:
  void io_error(char*);
  void begin_list(Tag);
  int  in_list();
  void end_list();
  int                   is_eol();
  int                   read_key();
  int                   read_tag(Tag);
  int                   read_int();
  Bits                  read_rootbits(int*);
  Bits                  read_bits(int);
  Pair<int,int>         read_pairint();
  Set<int>              read_setint();
  Set<Pair<int,int> >   read_setpairint();
};
```

```
// Modal output.  Mode is set when constructed.
class ModeOut : private ofstream
{
  ...
public:
  void indent();
  void exdent();
  void write_indent();

  void start_list();
  void incr_list();
  void finish_list();
  void set_eol();
  void write_key(int);
  void write_tag(Tag);
  void write_int(int);
  void write_rootbits(const Bits&);
  void write_bits(const Bits&);
  void write_pairint(const Pair<int,int>&);
  void write_setint(const Set<int>&);
  void write_setpairint(const Set<Pair<int,int> >&);
};
```

Each implementation *class* of ISM should provide a definition of the public `write` method, laying out non-process data in a way it can recover and simply using the `write` methods of any process data members it may have. In order to make the textual representation easier for humans to decipher, each "line" should start with a call to `write_indent()`. The global indentation level can be increased (before the [virtual] recursive calls to the ISM `write` method, for instance) by `indent()`, and reduced (on return) by `exdent()`; these are no-ops in binary mode.
In addition, it should define a private class-static method

`static ISMRef` *class*`::read(ModeIn &in);`

which reinterprets the non-process data and can use a (magic) call to `Fob::retrieve(in)` to recover each process member. In order to enable this technology, which provides link-time registration of new derived classes of ISM, the implementation must also define a class-static member of class Fob, and give its definition by a construction from the `read` method, thus:

`Fob` *class*`::fob ("`*name*`",` *class*`::read);`

where "*name*" is chosen to be mnemonic of "*class*" and unique among all implementations likely to be encountered (any duplication results in a load-time error).

4 Status and future developments

At the time of writing, **FDR 2** is not yet available as a fully integrated product. The infrastructure, core normalization and refinement functionality and many of the compression techniques form a library, against which appropriate `main` programs may be linked. The primary functions, corresponding to the **FDR** programs `normal` and `refine/refine2/refine2d/`etc, are currently supported as `fdr2norm` and `fdr2` respectively. Compression routines, too, have simple file-to-file interfaces.

A new module is available for incorporation into the existing Standard ML front-end engine `fdr`, which allows processes to be written to file in the new format. CSP "high-level" operators are by default mapped into structured files which give rise to hierarchical objects of the corresponding derived classes.

4.1 Debugging

In the case that a refinement check throws up a negative answer, `fdr2` yields a witness behavior which is a (minimal) counter-example to its truth. As with `refine`, this may be any one of three kinds:

- a divergence of the implementation which is not one of the specification;
- a non-divergent trace of the implementation which is not one of the specification;
- a trace of the implementation leading to a stable state where the minimal acceptances are not all (point-wise) supersets of any of those of the corresponding state of the specification (and so, in **FDR** terms, the maximal refusals are disallowed).

The difference between the **FDR 2** case and that of **FDR** is twofold:

- the state information (potentially on both machines, but particularly on the implementation) may have more complex structure than the (implicit) tupling of **FDR**'s state vectors
- on the other, there may well be structure underlying the implementation, which the user will want to analyse the erroneous behavior with respect to, which has been lost through compression or enumeration optimizations.

The additional decomposition (using virtual methods instantiated by the operators) and recalculation, respectively, of this information is supported within the **FDR 2** framework.

What remains is to provide a user interface to this debugging functionality. This most naturally moves from the Standard ML engine (which in **FDR** is responsible for decomposing the behavior across the entire "high-level" syntax tree, and supplying this information to the X11 graphical interface, but which is also planned to be obsolescent in the medium term, as discussed in the following section) into the graphical interface itself, with a more lazy, user-driven, approach.

This development is anticipated early in 1995.

4.2 Integration with parser/operational semantics engine

The outputs of the ONR-sponsored academic project at Oxford will go further than simply a parser and type-checker for the ASCII language. A tool with the ability to evolve the operational behavior of a process term is clearly a good candidate in the medium term for a replacement of the current Standard ML engine, `fdr`. In particular, we expect to be able to adapt it both to generate a closed-form of the "leaf" processes, and to "compile" the "supercombinator" representations referred to in Section 2, as well as more structured operator-oriented trees.

This development is anticipated in mid to late 1995.

5 Conclusion

Although somewhat delayed relative to internal plans, **FDR 2** shows every sign of being worth the wait. In particular, the care which has gone into developing clean interfaces and flexible infrastructure makes complex extensions extremely easy to program. For example, each of the following (required for other projects) took substantially less than a day to code:

- a program to calculate the set of events which label arcs from nodes which are actually reachable by a process (sometimes written $\sigma(P)$);
- an operator which normalizes a (potentially infinite) process lazily, as nodes in its normal form are first visited;
- a program to identify classes of nodes in the normal form which are visited in the course of a refinement check.

There are also a number of refinement problems which were outside the abilities of **FDR** which have submitted to the more structured approach of **FDR 2**.

References

[1] R.S. Bird and P. Wadler. *An introduction to functional programming.* Prentice-Hall, 1988.

[2] S.D. Brookes. *A model for communicating sequential processes.* D.phil., Oxford University, 1983.

[3] M.H. Goldsmith et al. *N00014-93-C-0213: Second Quarterly Report.* Technical report, Formal Systems Design and Development, Inc., P.O. Box 3004, Auburn, AL 36831-3004, 1994.

[4] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[5] D.M. Jackson and M.H. Goldsmith. Requirements for Refinement Checker Development. Technical report, Formal Systems Design & Development, Inc., April 1994. Deliverable D1.1 of SBIR N00014-93-C-0213, in [3].

[6] L. Jategoankar, A Meyer, and A.W. Roscoe. Separating failures from divergence. In preparation.

[7] A.W. Roscoe. Model-Checking CSP. In *A Classical Mind, Essays in Honour of C.A.R. Hoare.* Prentice-Hall, 1994.

[8] J.B. Scattergood. A basis for CSP tools. Oxford University Computing Laboratory Qualifying Dissertation, 1993.

FormalSystems

A CSP Priority Operator for FDR 2

Prototype Software for Discrete Real-Time Extensions to FDR

Michael Goldsmith

Formal Systems Design & Development, Inc.

December 16, 1994

Summary

This document describes some theoretical aspects of using an untimed but prioritized dialect of CSP to model discrete real-time behavior, and describes a prototype **FDR 2** operator definition (of the kind described in [1]) giving implementation to refinement over the new dialect.

1 Discrete real-time modeling

As discussed elsewhere ([6, 3, 5]), many of the problems with a naive approach to modeling the passage of time by the occurrence of some clock event (such as the tendency to detect pathological behavior where in fact none should arise, particularly where the system relies upon time-out constructs – since the symmetry of external choice allows the system to prefer the clock event causing the time-out even when the "desirable" events are also available) can be averted by judicous appeals to a notion of priority. Here we flesh out those notions.

Recall that, rather than the occam-style solution of a prioritized constructor (such as `PRI ALT`) to support the notion of priority and allow the programmer to force the system to "make progress" if possible, we are developing a theory which attaches priority to the events themselves: a (possibly partial) order between events such that the operational semantics of a process allows an event to occur only if all strictly higher priority events are not enabled. Progress has been made in refining the earlier commercial collaboration between Draper and Formal Systems (Europe) Ltd which tried out some of these notions (picking a set of "urgent" events to have priority over the clock ticks) with considerable success; the relation of an intuitive notion of the desired operational behavior with objects in the semantic space needed to reason about refinement were there entirely ad hoc and experimental, with little theoretical justification.

This is the textual component of Deliverable D1.2 in the Office of Naval Research SBIR Project N00014–93–C–0213 *Embedded Transputer-based System Design.*

As we have seen, the main obstacle to connecting the desired operational behavior to abstract semantic objects is that the standard semantic spaces are not sufficiently rich, at least for a denotational and compositional description. Among the algebraic properties which hold for untimed CSP is one that we may call *uncertainty*: that when, and how much, nondeterminism is resolved at any particular point in the evolution of a process is something which cannot be determined by observation. A concrete example of this is that the two processes

$$\begin{aligned} P &= (a \rightarrow Stop \,\square\, x \rightarrow x \rightarrow Stop) \sqcap x \rightarrow y \rightarrow Stop \\ Q &= (a \rightarrow Stop \,\square\, x \rightarrow y \rightarrow Stop) \sqcap x \rightarrow x \rightarrow Stop \end{aligned}$$

are equal, both behaving indistinguishably from

$$R \;=\; a \rightarrow Stop \,\square\, x \rightarrow (y \rightarrow Stop \sqcap x \rightarrow Stop)$$

If we are in a world where a events are preferred to x events, these three are operationally distinct: provided the environment is prepared to allow both events to occur, if an x happens it must be because the nondeterminism is resolved in favor of the second component (of P or Q; if a is not somehow blocked, R will always cooperate in taking the first branch); and their respective subsequent behavior is different. $P \parallel P$ can always (and only) do a y after an x (when it could have done an a) and $Q \parallel Q$ can always (and only) do a second x in the same circumstances; $P \parallel Q$ will deadlock after doing an x (if it could have done an a).

We remarked that the need for the qualifications concerning the availability of a yields the clue to our proposed solution. In the standard model, it does not matter what other events were allowed by the environment at the moment one happened; it is only at the point where none of the offered events are possible for the process that more than one is observed (as the refusal part of a failure). If a were not allowed by the environment at the start of the process, then all three processes again have indistinguishable behavior (as of the second component of R); that information is not available, however, in the data recorded in the event of an x being performed.

To rectify this, we record "historical" refusal information at each point along the trace: not only what events occurred, but which events were at that time declined, even if on offer from the environment[1]. In addition, we will still require "final" refusal information to record states which do not make progress on given offers.

This gives, for a given alphabet of discourse Σ, a semantic space

$$HT \subset \mathbb{P}((\Sigma \times \mathbb{P}\Sigma)^* \times \mathbb{P}\Sigma)$$

where each $P \in HT$ satisfies the following axioms:

$$(\langle\rangle, \emptyset) \in P \tag{1}$$

[1]This is analogous to the Timed CSP solution; that language, too, does not admit convexity and uncertainty laws because of the possibility of retracting offers of events, and the extended refusals of the timed models play a similar role in allowing the past behavior of the environment to be taken into account.

$$(s^\frown t, \emptyset) \in P \Rightarrow (s, \emptyset) \in P \tag{2}$$

$$(s, X) \in P \wedge Y \subseteq X \Rightarrow (s, Y) \in P \tag{3}$$

$$(s^\frown\langle(a, X)\rangle^\frown t, Z) \wedge Y \subseteq X \Rightarrow (s^\frown\langle(a, Y)\rangle^\frown t, Z) \in P \tag{4}$$

$$(s, X) \in P \wedge (s, X \cup \{a\}) \notin P \Rightarrow (s^\frown\langle(a, X)\rangle, \emptyset) \in P \tag{5}$$

$$(s^\frown\langle(a, X)\rangle, \emptyset) \in P \Rightarrow (s, X) \in P \tag{6}$$

The first three are identical to axioms of the standard failures-divergences model (although the types differ slightly); the fourth is a natural generalization of the third. Axiom 5 is the direct analogy of the standard one which ensures that any element of a minimal acceptance is also an available initial, but in addition mandates that appropriate "historical" observations can be recorded. Finally, Axiom 6 requires that "historical" information does indeed correspond to a possible history.

We have remarked before that enriching this space to contain a component for divergences causes no difficulty: we simply add a divergences component (of type $(\Sigma \times \mathbb{P}\,\Sigma)^*$) obeying precisely the standard axioms (modulo the type difference). So far, however, our expectation that, for the kind of systems we intend to model, the ability of (divergence-free) chaos to "stop the passage of time" suffices to catch that kind of pathology, has been positively reinforced by our experiences both in this project and elsewhere.

We are now in a position to present a semantics for a substantial subset[2] of the language:

$$\begin{array}{rcl} P & ::= & STOP \\ & | & e \rightarrow P \\ & | & P \,\square\, P \\ & | & P \sqcap P \\ & | & P \underset{E}{\|} P \\ & | & P \setminus E \\ & | & f^{-1}(P) \end{array}$$

(for e ranging over Σ, E ranging over $\mathbb{P}\,\Sigma$ and f ranging over finite-to-one mappings $\Sigma \rightarrow \Sigma$) into this model, allowing the desired distinction of the processes above.

$$\begin{array}{rcl} \mathcal{F}_{HT}[\![STOP]\!] & = & \{(\langle\rangle, E) \mid E \subseteq \Sigma\} \\ \mathcal{F}_{HT}[\![e \rightarrow P]\!] & = & \{(\langle\rangle, E) \mid e \notin E\} \cup \{(\langle e, E\rangle^\frown s, E') \mid e \notin E,\ (s, E') \in \mathcal{F}_{HT}[\![P]\!]\} \\ \mathcal{F}_{HT}[\![P_1 \,\square\, P_2]\!] & = & \{(\langle\rangle, E) \mid (\langle\rangle, E) \in \mathcal{F}_{HT}[\![P_i]\!],\ i = 1, 2\} \cup \\ & & \{(s, E) \mid s \neq \langle\rangle,\ (s, E) \in \mathcal{F}_{HT}[\![P_1]\!] \vee (s, E) \in \mathcal{F}_{HT}[\![P_2]\!]\} \\ \mathcal{F}_{HT}[\![P_1 \sqcap P_2]\!] & = & \mathcal{F}_{HT}[\![P_1]\!] \cup \mathcal{F}_{HT}[\![P_2]\!] \end{array}$$

[2]This is quite certainly sufficient, as the (finite) generalized operators and the other forms of parallel constructor are definable in terms of these constructs. Recursion is unproblematic.

$$\begin{aligned}
\mathcal{F}_{HT}[\![P_1 \underset{E}{\|} P_2]\!] \;=\;& \{(s,E') \mid \exists (s_i,E_i) : \mathcal{F}_{HT}[\![P_i]\!],\ \psi_i : \operatorname{dom} s_i \overset{m}{\rightarrowtail} \operatorname{dom} s \bullet \\
& \quad \forall k_i : \operatorname{dom} s_i \bullet \pi_1(s_i[k_i]) = \pi_1(s[\psi_i(k_i)]), \\
& \quad \forall j : \operatorname{dom} s \bullet \pi_1(s[j]) \in E \Leftrightarrow \\
& \qquad \exists k_i : \operatorname{dom} s_i \bullet \psi_1(k_1) = j = \psi_2(k_2), \\
& \quad \forall j : \operatorname{dom} s, k_i = \sup \psi_i^{-1}\{j' \mid j' \leqslant j\} \bullet \\
& \qquad \pi_2(s[j]) \cap E = (\pi_2(s_1[k_1]) \cup \pi_2(s_2[k_2])) \cap E, \\
& \qquad \pi_2(s[j]) \cup E = (\pi_2(s_1[k_1]) \cap \pi_2(s_2[k_2])) \cup E, \\
& \quad E' \cap E = (E_1 \cup E_2) \cap E, \\
& \quad E' \cup E = (E_1 \cap E_2) \cup E\} \\
\mathcal{F}_{HT}[\![P \setminus E]\!] \;=\;& \{(s,E') \mid \exists (s_0,E_0) : \mathcal{F}_{HT}[\![P]\!],\ \psi : \operatorname{dom} s \overset{m}{\rightarrowtail} \operatorname{dom} s_0 \bullet \\
& \quad \forall k : \operatorname{dom} s_0 \bullet k \in \operatorname{ran}\psi \Leftrightarrow \pi_1(s_0[k]) \notin E, \\
& \quad \forall j : \operatorname{dom} s \bullet \pi_1(s[j]) = \pi_1(s_0[\psi(j)]), \\
& \qquad\qquad \pi_2(s[j]) \cup E = \pi_2(s_0[\psi(j)]), \\
& \quad E' \cup E = E_0\} \\
\mathcal{F}_{HT}[\![f^{-1}(P)]\!] \;=\;& \{(s,E) \mid (f(\!|s|\!), f(\!|E|\!)) \in \mathcal{F}_{HT}[\![p]\!]\}
\end{aligned}$$

We can also define projections on this space: one, for any priority pre-order $\prec$ between events, gives the effect as if all events were always offered by the environment, thus finally resolving the affect of priority on choices (and reducing the "external" world to an observer – as much environment as necessary to control execution must be explicitly modeled and composed into the system):

$$\begin{aligned}
P^{\prec} \;=\;& \{(s,X) \mid \exists (s',X') : P \bullet \\
& \quad \pi_1 \circ s = \pi_1 \circ s', \\
& \quad \forall j : \operatorname{dom} s \bullet \pi_2(s[j]) \cup \{a' : \Sigma \mid \pi_1(s[j]) \prec a'\} = \pi_2(s'[j]), \\
& \quad \forall a : X \bullet (s', X' \cup \{a' : \Sigma \mid a \prec a'\}) \in P \Rightarrow a \in X'\}
\end{aligned}$$

A second projection obscures all interesting refusal information in the traces:

$$\begin{aligned}
\overline{P} \;=\;& \{(s,X) \mid \forall t < s \bullet \exists (t',X') : P\, ;\, a : \Sigma \bullet \\
& \qquad \pi_1 \circ t = \pi_1 \circ t' \wedge t'^\frown\langle(a,X')\rangle \leqslant s \\
& \quad \exists (s',X') : P \bullet \\
& \qquad \pi_1 \circ s = \pi_1 \circ s' \wedge X = X'\}
\end{aligned}$$

The effect of this is to close up under uncertainty, and the resulting retract is naturally isomorphic (under Φ, say) to the standard failures model (simply by stripping the "historical" refusals from the traces entirely). Since this is clearly monotonic, if refinement holds in the richer model then it will hold between images in the standard model, and any failure to refine there will reflect a failure in the desired space; we can thus use the existing check as a partial test for the overall result. The uncertainty retraction maps processes onto the weakest one with the same set of everywhere-empty-historical-refusal failures; thus when the desired specifications are themselves "uncertain" the induced check is entirely accurate.

There are some unpleasant (but not unexpected – it has long been established that discrete-time analysis fits ill with compositionality [7]) interactions between these projections and the CSP operators (promoted to the semantic space). In particular, we do not have any of the following holding in general in the HT model:

$$\begin{aligned}
\overline{P \setminus E} &= \overline{P} \setminus E \\
\overline{(P \setminus E)^{\prec}} &= \overline{P^{\prec}} \setminus E \\
\overline{(P \underset{E}{\|} P')^{\prec}} &= \overline{P^{\prec}} \underset{E}{\|} \overline{P'^{\prec}} \\
\overline{(P \underset{E}{\|} P')^{\prec}} &= \overline{(\overline{P^{\prec}} \underset{E}{\|} \overline{P'^{\prec}})^{\prec}}
\end{aligned}$$

The last result does hold, however, when all the elements of the synchronization set are $\prec$-minimal. Thus events which cannot affect the availability of others behave almost normally across interfaces of composition.

2 Implementation for FDR 2

We are encouraged by this last result to put these ideas into practice, exploiting the virtues of flexibility, extensibility and (effectively) rapid prototyping provided by the **FDR 2** infrastructure. The obvious candidate for a first useful "operator" (in the sense of [1]) is $(P, \prec) \mapsto P^{\prec}$; where provided P is given as a pure transition system where all τs arise from explicit nondeterminism, not hiding, the same is true of the result and in both cases the "historical" behavior can be deduced from its operational semantics.
In order to achieve this, we derive a new class from the abstract ISM interface (see [1, §3]):

```
class OpPrioISM : public ISM
{
private:
  ISMRef p;
  Mapping< Event, Set<Event> > rho;

  Set<Event> restrict_by_set(const Set<Event> &inhibit,
                             const Set<Event> &query) const;
  Set<Event> restrict_by_node(const Node &n, const Set<Event> &query) const;
  Set<Event> necessary(const Node &n) const;

  static ISMRef read(ModelIn &);
  static Fob fob;

  OpPrioISM(const Mapping< Event, Set <Event> > &r,
            const ISMRef &p);
public:
  Set<Event> alphabet() const;
  Node root() const;
  Set<Event> initials(const Node &) const;
  Set< Set<Event> > minAcceptances(const Node &) const;
  Set<Node> afters(const Node &, const Event &) const;
  int divergent(const Node &) const;
  int markedDivergent(const Node &) const;

  void write(ModeOut &) const;
};
```

Of the private members, `p` and `rho` store the “arguments” of the operator[3] ($a \prec b$ iff `rho.apply(a).member(b)` is non-zero); `read` and `fob` are the required I/O ancillaries. The constructor is currently made available only through the `read` method. The other private methods assist the definition of the simulation interface methods.

2.1 Auxiliary functions

The first, `restrict_by_set`, subtracts from its second argument any visible events which are `rho`-dominated by any element of its first:

```
Set<Event> OpPrioISM::restrict_by_set (const Set<Event> &inhibit,
                                       const Set<Event> &query) const
{
  if (query.empty () || inhibit.empty ())
    return query;
  Set<Event> result;
  for(Iter< Event > i=query; i.going(); i.next())
  {
    Event e = i.here ();
    if (is_tau (e))
      result.add_singleton (e);
    else
    {
      if (setinter(inhibit, rho.apply(e)).empty())
        result.add_singleton (e);
    }
  }
  return result;
}
```

[3]`Mapping<S,T>` is a simple templated class representing finite functions, with methods for functional override (`add`), domain membership (`inDom`) and extent (`dom`), and application (`apply`).

The events which are **necessary** for a node are the interior (intersection) of its minimal acceptances:

```
Set<Event> OpPrioISM::necessary (const Node &n) const
{
  Set< Set<Event> > mas=minAcceptances (n);
  Set<Event> result;
  if (mas.empty ())
    return result;
  Iter < Set<Event> > i=mas;
  for (result=i.here (), i.next();
       (!result.empty ()) && i.going ();
       i.next ())
    result = setinter (result, i.here ());
  return result;
}
```

and these are precisely those which can prevent transitions from contributing to the initials of a node:

```
Set<Event> OpPrioISM::restrict_by_node (const Node &n,
                                        const Set<Event> &query) const
{
  if (query.empty ())
    return query;
  else return restrict_by_set (necessary (n), query);
}
```

2.2 Simulation interface

Adding priority does not change the universe of events under consideration or the distinguished starting point of the transition graph, so these are passed on from the argument process:

```
Set<Event> OpPrioISM::alphabet (void) const
{
  return (p->alphabet ());
}

Node OpPrioISM::root (void) const
{
  return (p->root ());
}
```

The possible initials of a node are precisely those which are not inhibited by the events which are impossible to refuse:

```
Set<Event> OpPrioISM::initials (const Node &n) const
{
  Set<Event> inits = p->initials(n);
  return restrict_by_node (n, inits);
}
```

whereas each minimal acceptance inhibits itself (for if a low-priority and a high-priority event are both "irrefusible" in the unprioritized world, then the low-priority one will never occur, let alone be irrefusible, in the prioritized case):

```
Set< Set<Event> > OpPrioISM::minAcceptances (const Node &n) const
{
  Set< Set<Event> > result, mas=p->minAcceptances (n);
  if (mas.empty ())
    return mas;
  Set<Event> ma;
  int count=0;
  for (Iter< Set<Event> > masi=mas; masi.going(); masi.next())
  {
    ma = masi.here ();
    if (ma.empty ())
      return mas;
    ++count;
    result.add_singleton (restrict_by_set (ma, ma));
  }
  return count > 1 ? upward_open (result) : result;
}
```

The final conditional here takes account of the fact that (incomparable) minimal acceptances of the argument may be mapped to comparable sets; `upward_open` removes any elements of its argument which are proper supersets of another element.

The final interesting case is calculating the **afters** of a node. This simply follows the intuition that we are removing transitions which are inhibited; that is, those whose labels are no longer in the initials:

```
Set<Node> OpPrioISM::afters (const Node &n, const Event &e) const
{
  Set<Event> inits=initials(n);
  if (inits.member (e))
    return p->afters (n, e);
  else
    return empty_node_set;
}
```

3 Conclusions

The denotational model appears quite complex, but this captures what is in fact a very simple intuitive concept over the operational semantics of CSP. Since the basic modeling of processes within **FDR 2** is at the operational level, this means that the definition of the prioritization operator is reasonably straightforward.

There are some caveats which apply, however, due to the fact that the operator definition essentially gives a modified operational description to an operationally presented process:

- The operator is therefore sensitive to transformations of its argument which are congruences with respect to the standard denotational semantics. In particular, normalization will close up under uncertainty; and many compression techniques will also do so in whole or in part. (Simply rendering the transition graph as an explicit tabulation of the states in order to optimize access to the semantics *is* safe, of course.)

- The same considerations apply after the operator has been applied. Any further processing may increase or decrease the uncertainty, but this is unproblematic because of the nature of that processing.

- Any refinement check is (necessarily, in the current and any conceived future framework) against a normalized specification. Normalization closes under uncertainty, and yields a transition graph which can sensibly only be identified with the minimal process under $\sqsubseteq_{HT}$ sharing the same event-traces and (final) refusals. The degree of uncertainty of the implementation side cannot affect the result of the check. We have not therefore attempted to implement the closure under uncertainty operationally (which would be possible, if unproductive).

Specifications which do not exploit any relative prioritization of the events they may engage in clearly still make sense here: among these are commonly useful properties such as deadlock freedom. Trace specifications also correspond to using the most nondeterministic process with the desired trace set, and so may be used freely in this context. Specification idioms which do attempt to exploit priority are a topic which requires further investigation.

References

[1] P.H.B. Gardiner and M.H. Goldsmith. Inside **FDR 2**. Technical report, Formal Systems Design & Development, Inc., 1994. Adjunct to D1.2 of SBIR N00014-93-C-0213, in [4].

[2] M.H. Goldsmith et al. *N00014-93-C-0213: Second Quarterly Report.* Technical report, Formal Systems Design and Development, Inc., P.O. Box 3004, Auburn, AL 36831-3004, 1994.

[3] M.H. Goldsmith et al. *N00014-93-C-0213: Third Quarterly Report.* Technical report, Formal Systems Design and Development, Inc., P.O. Box 3004, Auburn, AL 36831-3004, 1994.

[4] M.H. Goldsmith et al. *N00014-93-C-0213: Fourth Quarterly Report.* Technical report, Formal Systems Design and Development, Inc., P.O. Box 3004, Auburn, AL 36831-3004, 1994.

[5] David M. Jackson and Richard O. Chapman. Models of the Fault-Tolerant Processor; Architecture and Verification. Technical report, Formal Systems Design & Development, Inc., 1994. Deliverable to SBIR N00014-93-C-0213, in [4].

[6] D.M. Jackson and M.H. Goldsmith. Scheduler Specification in CSP: Fixed Priority Preemptive Example. Working Paper W.2.2.1, Formal Systems Design & Development, Inc., April 1994. Included in Report of SBIR N00014-93-C-0213, in [2].

[7] G. Jones. *A Timed Model of Communicating Processes.* D. Phil. thesis, Oxford University, 1982.

This is Release 1.0 of this document, last modified by Michael Goldsmith at 17:49:26 GMT on December 16, 1994.

FormalSystems

Models of the Fault-Tolerant Processor Architecture and Verification

David Jackson
Richard Chapman

December 16, 1994

Summary

This document describes our work to date on formalizing the design and analysis of the Transputer Fault-Tolerant Processor system. The early sections summarize the fault-tolerance properties which we intend to verify, our model and a simple demonstration that the architecture does meet our requirement for Byzantine fault-tolerance. We then describe how such verification can be simplified if we exploit the symmetry both the overall design, and of the behavior of its components. The next section describes how we can relate these models of network behavior to the application level scheduling problem, and in particular how we can exploit temporal redundancy to tolerate transient faults. It includes discussions on voting, permuted schedules and on transient recovery techniques. Abstract models of task execution and voting are given which, despite their simplicity, provide a framework for future models of specific scheduling policies. We include two more detailed models of the system which analyse a distributed model of the system using synchronous and partly asynchronous models.

Contents

1 Fault-Tolerant Behavior

We begin by summarizing the types of behavior which we ultimately intend to analyse in order to show how they can be expressed in terms of the models we will describe below.

1.1 Classes of Fault-Tolerance

The first class of faults we will consider, and the errors they may cause are, those outlined in a previous project document, [2]. These are the *Byzantine* failures of a system element, after which no assumptions can be made about the behavior of the component. In particular, such faults may not be manifest – failure may not be obvious to those parts of the system with which the failed element interacts. It is well-established that a suitably redundant system can tolerate Byzantine failures, but that the cost of such a system is higher (i.e. it requires greater redundancy) than a system designed to tolerate manifest faults (see, for example, [8]).

We can exploit this redundancy, however, to improve tolerance to other types of fault. Of particular importance are "common-mode" errors which arise in a number of replicated elements simultaneously, perhaps as the result of some environmental factor. Where these errors arise from transient faults (such as corruption of semiconductor memory) we can use *temporal* redundancy to allow correct operation to be resumed even if the number of elements affected is much greater than the number of Byzantine failures that a system might tolerate. This strategy has again been outlined in a previous project report [1].

Obviously these two situations are far from being an exhaustive catalogue of fault situations which we might design a system to tolerate but they do represent a possible extremes: in the Byzantine case we suffer complete non-manifest failure of few components, in the transient case we tolerate identifiable temporary faults in many. Other combinations, such as manifest permanent faults, may be included in later analysis.

1.2 Fault models

Modeling a component capable of Byzantine failure is relatively straight-forward, because we need to satisfy very few constraints on behavior after an error, but we must nevertheless take into account the features that our model represents if we are to provide a satisfactory model.

High level abstractions In models at the highest level of abstraction (the replicated synchronous view of [2]), a failed component can be represented as ignoring all inputs. We choose to ignore, rather than to refuse, inputs in order to remove the need to model details of the error detection and buffering which is used in practice to implement communication between distributed components. These communication elements are, of course, modeled in the lower level abstractions (Section 6 of this report). Abstracting from the implementation of the communication and error detection mechanism also influences the way we should model outputs from a faulty system element. The most obvious approach is to allow arbitrary generation or refusal of output events. This correctly captures the idea that a failed component exhibits the most general possible behavior but does not reflect the ability of a receiver to detect when outputs are being refused (typically by means of a time-out). We therefore model a faulty output as a combination of arbitrary valid outputs together with a distinguished error value which is always potentially available. While placing constraints on faulty behavior may appear unrealistic, it should be remembered that our fault model is actually also incorporating a significant amount of information about the ability of connected components to detect faults. We will make this information more explicit in later sections, but this abstract model will remain useful *because* it does not make assumptions about how communication errors are detected, and thus applies to a wide range of possible error detection techniques, including time-outs, parity or check-sum errors or more complex protocols.

Lower level abstractions In more detailed models the models of faulty components actually become simpler, because we are able to model more faithfully the way in which errors are detected by the remainder of the system. Both input and output behavior of a process after the occurrence of a Byzantine fault can be assumed to be entirely arbitrary: both inputs and outputs can be performed in any order, or refused at any stage. This is exactly the behavior of the $CHAOS$ process of CSP, as we might expect of a completely undetermined behavior.

By their very nature, transient faults require a more detailed model of the internal state of a system than Byzantine failure. The essence of our approach will be to decompose the application calculations into a series of tasks each of which calculates new values for part of the system state (and may produce outputs) from the previous system state and any inputs present. A transient fault is modeled by assuming that the fault corrupts some part of the processor's state arbitrarily, and that all tasks depending on that part of the state may in turn corrupt their outputs and final states. The task of the fault management system is to identify the corrupted parts of the system state and re-generate it where possible. Adding sufficient information to our high-level model to support this reasoning is discussed in Section 4 and later sections.

2 High-level Architecture

For practical applications, we will assume that tolerance of a single Byzantine fault is sufficient, and thus we will concentrate on quad-redundant systems. Each of the four redundant *fault-containment regions* (FCRs) which make up such a system must execute both the application tasks and the functions related to fault management: in our demonstrator application each FCR will typically contain two processors, one executing the application and another managing communication and input-output. This bipartite view is also applicable to single processor systems built using Transputer hardware, as separation between processing and communication is present even if the components are actually a CPU and a link engine on a single IC.

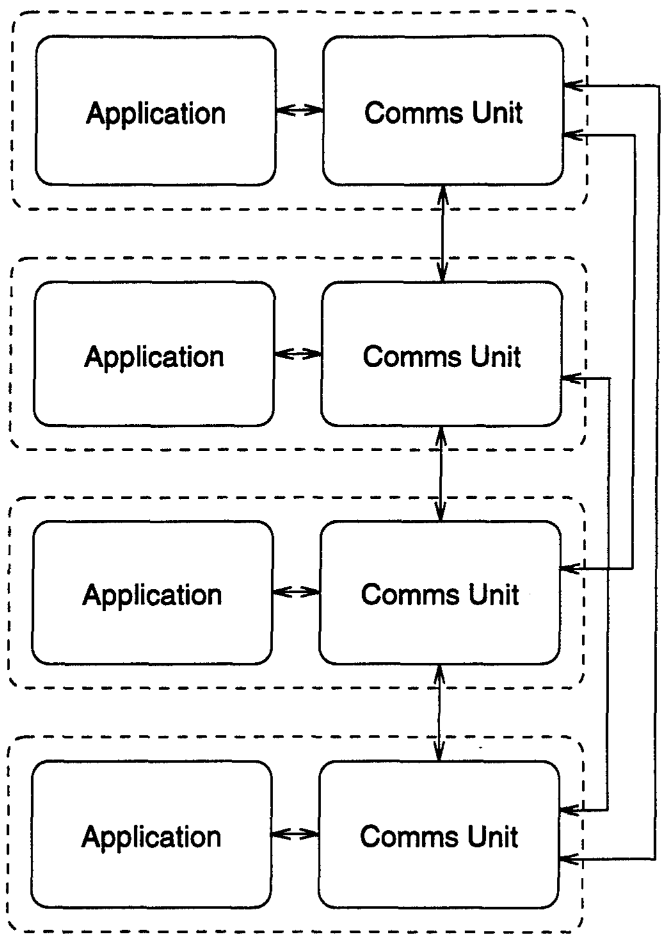

Figure 1: Communication between peers

2.1 Implementing the Oral Messages algorithm

As discussed in [2], we will use the *Oral Message* (OM) algorithm to establish consensus values for data in the presence of faults. Each FCR will communicate its local values for state and output data to its peers, and vote upon a derived value using its local data and the values it receives in return. The communication will have the pattern shown in Figure 1. Each node in Figure 1 represents the communications processing element of an FCR. Data is received from the application along the *in* channel and passed out along the cross channels (the vertical links in the diagram). Values received, along with the original value received, are combined by a majority voting process and the result is passed to the application or the environment.

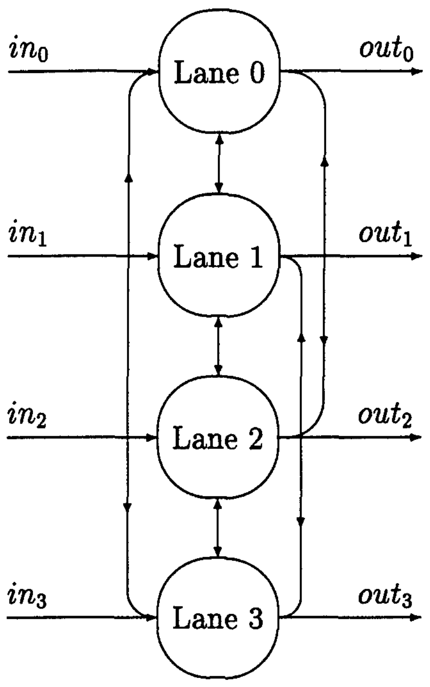

Figure 2: High-level Model Architecture

3 Tolerance of Byzantine Faults

We start our analysis with a high-level abstraction which serves to justify our primary claim of tolerance to Byzantine faults. The following model has the same structure as that outlined in Section 2.1. We concentrate on modeling the communication behavior of the system, and thus model the behavior of the input-output subsystem alone, representing data flowing to or from the application or IO devices by sets of channels *in* and *out*. The pattern of communication is then as shown in Figure 2. The channels *in* and *out* may not, of course, exist as explicit data paths in the case that application processing and communications are combined on a single processor, but there will always be some identifiable transfer of data corresponding to them. Each FCR (i.e. each node in Figure 2) is represented by two processes, one representing the outward transfer of local data to peer FCRs, the other representing voting using data received.

The desired behavior of our system is described in [2] in terms of two properties of a system distributing data from a single source by means of a two-stage algorithm. The properties are:

Agreement If two processors are non-faulty, they agree on the data values which they believe are being communicated.

Validity If the originator of a data item is non-faulty, all non-faulty processors derive the correct value.

To model the two stage transmission we will consider a network consisting of the four communication elements of our system together with an addition process which performs the initial data distribution. In a physical system we would expect this additional task to be implemented within one particular FCR, the *transmitter* of the data flow being considered, while the other FCRs would be *receivers* of the flow.

In order to capture the *validity* property described above, we add a further component to our network which does not correspond to any actual implementation process, but rather captures our ability to observe the system: if each FCR delivers the value it computes to a final overall majority vote, then if the validity property holds of outputs of the FCRs, the output of the voter must always match the value provided by the data source. The overall data-flow through the network is shown in Figure 3. We require that this complete system, when viewed as a data transmission medium between its source and the final output, is a perfect buffer, provided that the first-round data distribution is non-faulty. This must hold even if one of the receiver FCRs is Byzantine faulty. A CSP model of this system (suitable for analysis with the **FDR** [3] tool) is given below.

tftp.csp: Model demonstrating tolerance of 4-FCR Oral-Messages algorithm to a single Byzantine fault.

(c) Formal Systems Design & Development, Inc, 1994

Originated by: Dave Jackson.

```
-- This version: $Id: tftp.csp,v 2.0 1994/12/16 17:44:03 dave Del $
```

In the current model we are principally interested in the distinction between a real data value and a potentially erroneous one. It will suffice, for the present, to consider a single "good" data value, and an error token, Err:

```
RAWDATA = {0}
```

`Err = 99` *Any value not in RAWDATA*

```
DATA = union({Err},RAWDATA)
```

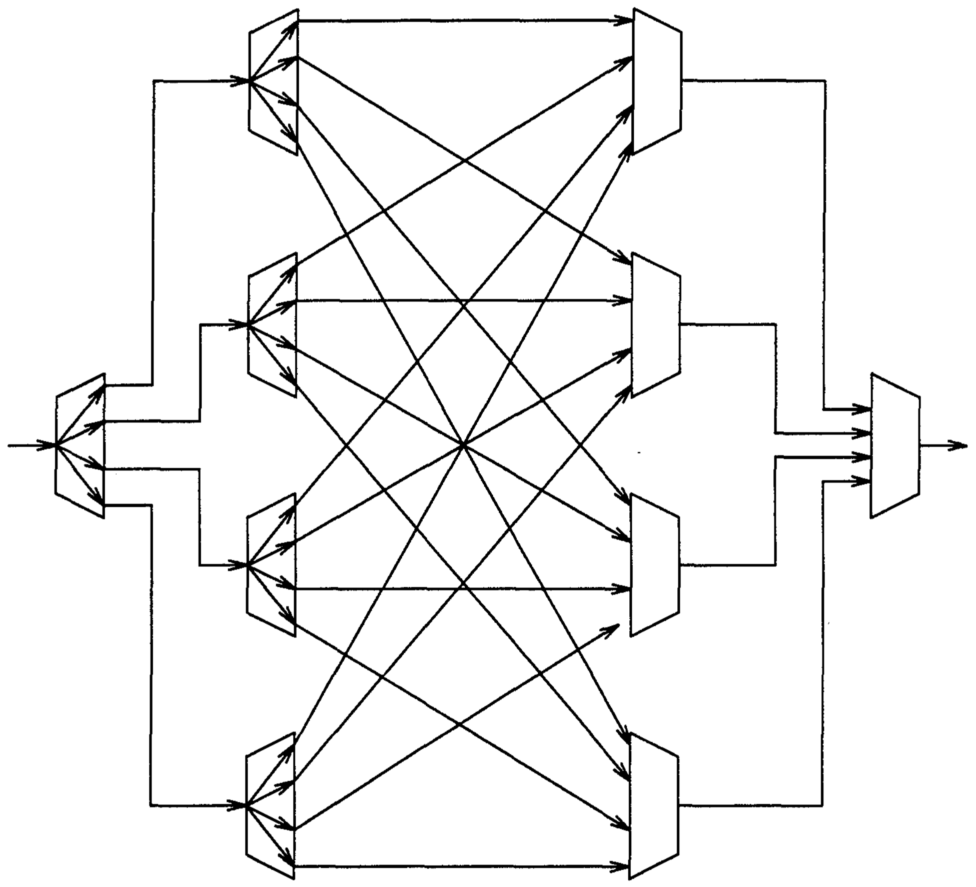

Figure 3: Detail of Data-flow Through FTP Model

Specification of data exchange mechanism

We require that data is transmitted from the data source to the outputs of each communications element in such a way that a majority vote over all those outputs correctly reflects the input. Our high level specification is thus that the system is a buffer. We can in fact show that the system represents a deterministic buffer, as follows:

The initial state of an n place buffer is empty:

```
BUFFER(n) = BUFF(<>,n)
```

For any state of the buffer, if it is empty it must accept an input.

```
BUFF(t,n) =
  if (null(t)) then source?x:RAWDATA -> BUFF(<x>, n)
  else
```
Otherwise, if the buffer is full it offers only an output.
```
if ((#(t))==n) then sink!(head(t)) -> BUFF(tail(t), n)
```
The final case, where the buffer is neither full nor empty allows both input and output.
```
else            (source?x:RAWDATA -> BUFF(t^<x>, n))
                    []
                     (sink!head(t) -> BUFF(tail(t), n))
```

At the time of writing, the FDR tool requires that we specify a fixed maximum size for our specification:
```
BUFF4 = BUFFER(4)
```
(This restriction is not theoretically necessary and we expect to be able to relax this constraint in future versions of FDR2.)

Model of OM Algorithm for Four FCRs

The most complex basic component in the algorithm is the voting module: the following process takes inputs from the channels specified in the set sources and passes majority voted values to the channel sink

Voting is encoded by maintaining sets of those channels which have supplied values for each data type, including error returns. Initially these sets are empty:

```
MAJ(sources, sink) = MAJORITY(sources, {}, {}, {}, sink)
```

While it is accepting input, the voter offers a choice over the inputs which it is still expecting to receive. When input is received, the channel is added to the appropriate set, and removed from the set of exepected inputs.

```
MAJORITY(expected, zeroes, ones, errs, sink) =
  if (card(expected) == 0) then OUTPUT(zeroes, ones, errs, sink)
    else ([] x : expected @ (x?y ->
      (if (y==0)
        then MAJORITY(diff(expected,{x}),
                      union(zeroes,{x}), ones, errs, sink)
        else if (y==1) then
             MAJORITY(diff(expected,{x}),
                      zeroes, union(ones,{x}),errs,sink)
        else
             MAJORITY(diff(expected,{x}),
                      zeroes, ones,union(errs,{x}),sink))))
```

When all expected inputs have been received, the voter supplies an output according to the size of the sets of inputs received. (For a single element data domain, we output the same (valid) value for any combination of valid inputs.)

```
OUTPUT(zeroes, ones, errs, sink) =
  if ((card(zeroes)==4) or (card(zeroes)==3))
    then sink!0 -> MAJ(Union({zeroes, ones, errs}), sink)
    else sink!0 -> MAJ(Union({zeroes, ones, errs}), sink)
```

The other required component is a data distribution process. While we could write this in a sequential form similar to the voter, we feel the symmetry of the action is made clearer if we express this process as a parallel composition of simple buffers. These buffers synchronize on their input but not on their output, yielding the required interleaving behaviour.

```
COPY = inp ? x:RAWDATA -> o0 ! x -> COPY
```

The following channel definitions specify the input to, and outputs from the first-stage data distribution. Later instances of the data distribution process will be derived by renaming this first one:

```
pragma channel inp : DATA
pragma channel o0, o1, o2, o3 : DATA
```

```
INSERT =  COPY [| {inp } |]
          ((COPY[[ o0 <- o1 ]]) [| { inp } |]
          ((COPY[[ o0 <- o2 ]]) [| { inp } |]
          (COPY[[ o0 <- o3 ]])))
```

The following channels define the external interfaces to our model:

```
pragma channel source, sink : DATA
```

and these implement the connections between peers:

```
pragma channel xcmid : DATA
pragma channel xc01, xc02, xc03 : DATA
pragma channel xc10, xc12, xc13 : DATA
pragma channel xc20, xc21, xc23 : DATA
pragma channel xc30, xc31, xc32 : DATA
```

and finally, the channels which represent the input and output from each of the FCRs:

```
pragma channel ain, bin, cin, din : DATA
pragma channel aout, bout, cout, dout : DATA
```

For brevity in later descriptions, we define sets of channels representing the inputs:

```
XCI0 = {xc10, xc20, xc30}
XCI1 = {xc01, xc21, xc31}
XCI2 = {xc02, xc12, xc32}
XCI3 = {xc03, xc13, xc23}
```

and outputs

```
XCO0 = {xc01, xc02, xc03}
XCO1 = {xc10, xc12, xc13}
XCO2 = {xc20, xc21, xc23}
XCO3 = {xc30, xc31, xc32}
```

connecting each FCR to its peers. The total interface sets of each FCR are as follows:

```
ALPHAA = (Union({{ain, aout}, XCI0, XCO0}))
ALPHAB = (Union({{bin, bout}, XCI1, XCO1}))
```

```
ALPHAC = (Union({{cin, cout}, XCI2, XCO2}))
ALPHAD = (Union({{din, dout}, XCI3, XCO3}))
```

We may now define processes representing each FCR. Each consists of a data distributor communicating with a voter by a channel xcmid. The data distributor also provides outputs XCOn and the voter accepts inputs from set XCIn.

```
FTLANEA =
  ((INSERT [[inp<-ain, o0<-xcmid, o1<-xc01, o2<-xc02, o3<-xc03]])
  [(union({ain, xcmid}, XCO0))||(union({aout, xcmid}, XCI0))]
  (MAJ(union(XCI0,{xcmid}), aout)))
  \ {xcmid}

FTLANEB =
  ((INSERT [[inp<-bin, o0<-xcmid, o1<-xc10, o2<-xc12, o3<-xc13]])
  [(union({bin, xcmid}, XCO1))||(union({bout, xcmid}, XCI1))]
  (MAJ(union(XCI1,{xcmid}), bout)))
  \ {xcmid}

FTLANEC =
  ((INSERT [[inp<-cin, o0<-xcmid, o1<-xc20, o2<-xc21, o3<-xc23]])
  [(union({cin, xcmid}, XCO2))||(union({cout, xcmid}, XCI2))]
  (MAJ(union(XCI2,{xcmid}), cout)))
  \ {xcmid}

FTLANED =
  ((INSERT [[inp<-din, o0<-xcmid, o1<-xc30, o2<-xc31, o3<-xc32]])
  [(union({din, xcmid}, XCO3))||(union({dout, xcmid}, XCI3))]
  (MAJ(union(XCI3,{xcmid}), dout)))
  \ {xcmid}
```

The fault-tolerant communication system as a whole is a parallel combination of these:

```
FTBUFF =
  (((FTLANEA [ALPHAA||ALPHAB] FTLANEB)
  [union(ALPHAA, ALPHAB)||union(ALPHAC,ALPHAD)]
    (FTLANEC [ALPHAC||ALPHAD] FTLANED)) \
                Union({XCI0, XCI1, XCI2, XCI3}))
```

The following sets define the interfaces of the first-level data distribution, the voting *network just defined, and the majority voter used to complete the model.*

```
ALPHAIN = {source, ain, bin, cin, din}
ALPHAFT = {ain, bin, cin, din, aout, bout, cout, dout}
ALPHAMJ = {sink, aout, bout, cout, dout}
```

These components are combined as follows:

```
SYSTEM =
  (((INSERT [[inp<-source, o0<-ain, o1<-bin, o2<-cin, o3<-din]])
  [ALPHAIN||ALPHAFT]
  FTBUFF)
  [(Union({ALPHAIN, ALPHAFT}))||ALPHAMJ]
  MAJ({aout, bout, cout, dout}, sink)) \ ALPHAFT
```

We hope, and indeed find, that $BUFF4 \sqsubseteq SYSTEM$

Now consider a failed processor, assumed not to be the source of single source data:

```
RUN(A) = [] a:A @ a -> RUN(A)

FTLANED' = RUN(Union({events(i)| i<- union({din},XCI3)}))
           ||| CHAOS(Union({events(i) | i <- union({|dout|},XCO3)}))
           ||| RUN({c.Err | c <- union({dout},XCO3)})
```

NB broken channel always allows error outputs.

```
FTBUFF' =
  (((FTLANEA [ALPHAA||ALPHAB] FTLANEB)
  [union(ALPHAA, ALPHAB)||union(ALPHAC,ALPHAD)]
    (FTLANEC [ALPHAC||ALPHAD] FTLANED')) \
                Union({XCI0, XCI1, XCI2, XCI3}))

SYSTEM' =
  (((INSERT [[inp<-source, o0<-ain, o1<-bin, o2<-cin, o3<-din]])
  [ALPHAIN||ALPHAFT]
  FTBUFF')
  [(Union({ALPHAIN, ALPHAFT}))||ALPHAMJ]
  MAJ({aout, bout, cout, dout}, sink)) \ ALPHAFT
```

We observe (for a single element data set + error value) that $BUFF4 \sqsubseteq SYSTEM'$

(approx 2.4M state pairs).

4 Refining the Architecture

4.1 Exploiting Symmetry

Whether verification is carried out by hand or mechanically, analysis of detailed models of the fault-tolerant processor will involve significant effort. From the structure of the model, however, we can see a very clear symmetry between the four components of the network.

We can exploit this symmetry in a number of ways, the most obvious being a reduction in the number of possible failures to be considered. If the network is operating in a fully symmetric manner, it obviously does not matter which of the processors is considered to fail, and we may thus isolate failures to an arbitrary fixed FCR. If the operation is asymmetric, as in the distribution of single-source data, we way still exploit the three-fold symmetry of the receiver processes and model single faults by only two cases: either the transmitter fails, or one of the receivers does. In the latter case the identity of the failed lane can again be arbitrary.

A more powerful technique exploits not only the large-scale symmetry of the network, but also takes advantage of the symmetric behavior of the components. Suppose, for example, that the data distribution process is entirely symmetric as regards the order and manner in which it makes its output available, as is the case for the process *INSERT* in the model of Section 3. Recall the data-flow shown in Figure 3; if we concentrate on the values produced by any one of the local voters, we see that it depends on only four of the data interchanges (i.e. the value from its local input and values received from its three peers. The relevant paths are highlighted in Figure 4. The behavior of each data distribution phase when we consider only one of its outputs will be significantly simplified, and indeed in many cases[1] it will degenerate to a simple form of buffer. The overall system which we must analyse to predict the output of a given voter reduces to that shown in Figure 5. We can use this simplification to allow us to prove properties of the whole system by considering only a single voter. Suppose we show that the output of the voter in the figure agrees with its input provided that no more than one of the preceding buffers (representing the data distribution operation of each FCR) is faulty. Unless we use explicit assumptions about which voter we consider and which buffer is faulty, our reasoning must then be valid for any functioning voter, and any single failure: we have established that *all* functioning voters agree with the input. In practice, of course, not all the data distributors will be identical in their relationship to the voter, because one of them

[1]Typically those where blocking one output does not prevent further inputs and outputs on other channels.

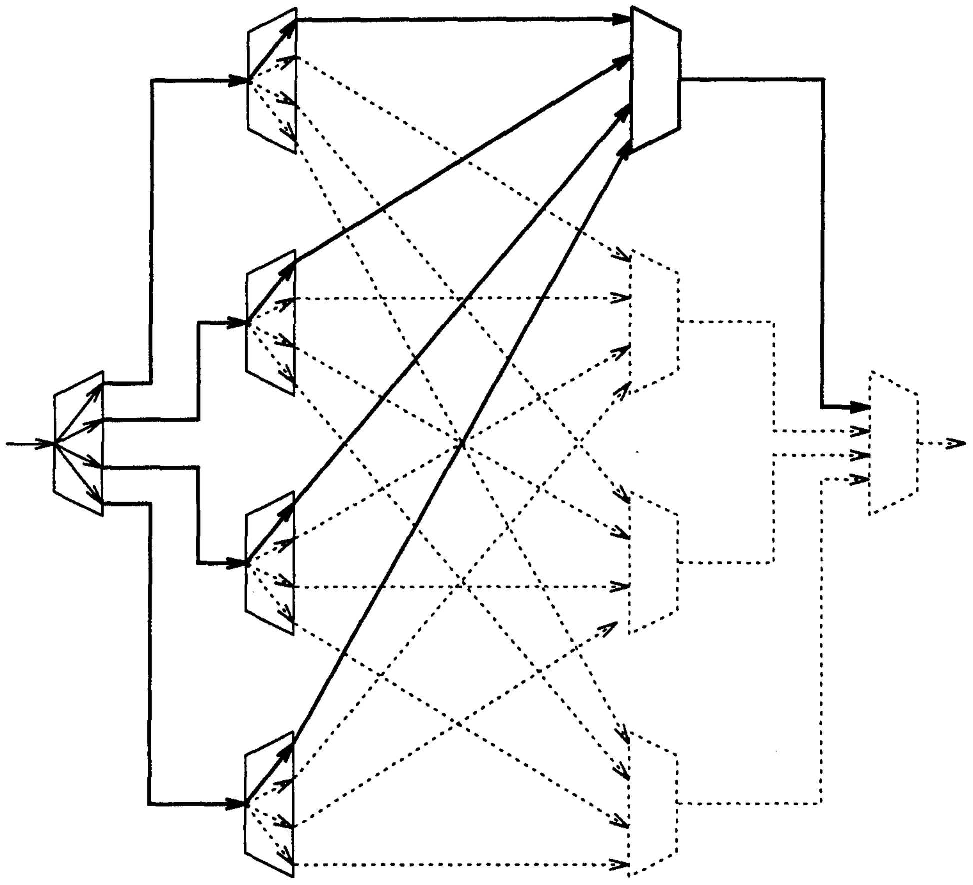

Figure 4: Data-flow to a Single Voter

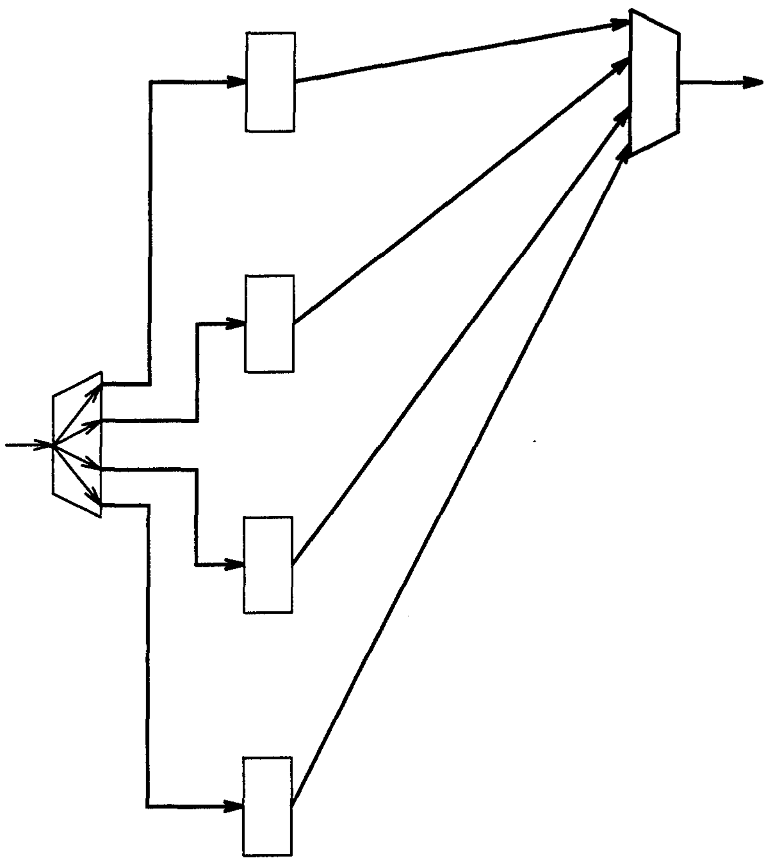

Figure 5: Analysing the Output of a Single Voter

will be contained in the same FCR, and may be implemented by the same processor. However, presence of a Byzantine fault in this distributor will then imply a potential fault in the voter, and we do not need to (and cannot expect to) establish that FCRs behavior.

This approach allows our models to concentrate on the behavior of a single component in our system, rather than having to model and analyse four identical replicas. We shall use this technique extensively in the following sections.

5 Fault-management & Application Programs

We have demonstrated in the preceding sections that tolerance to Byzantine faults can be achieved by designing our network of replicated processors to implement the Oral Messages algorithm. This tolerance is a property of the network and its communication pattern, and is thus independent of the actual application program, provided that sufficient data is exchanged and voted upon to keep the replicated copies of the program in agreement.

The second goal of our approach is to tolerate transient faults, including (but not limited to) those which affect a large proportion of our network for a brief interval. Designing and verifying strategies to achieve this aim will necessarily involve a more detailed knowledge of the operation of the application program than we have used in the earlier parts of this document. In particular, we will need knowledge of the tasks executed by the application program and their data dependency and scheduling constraints. We will adopt the view that real-time applications are typically constructed as a set of atomic tasks, exchanging data by means of shared variables, and subject to data-dependency and timing constraints as discussed in [7].

5.1 Permuted scheduling

One method identified in [1] to reduce the impact of multi-processor transient faults is to ensure that our replicated processors execute different tasks at each instant: rather than fix a schedule for executing application code, we define a set of *permuted schedules*. We intend that if a transient fault disrupts the tasks executing at a given time on a number of processors, then there should still be enough redundant executions of those tasks completed at other points in the same scheduler cycle for valid results to be obtained by voting.

5.1.1 Validity of Permutation

The potential benefits of permuted schedules can be verified using a relatively straightforward, if potentially unwieldy CSP model. The model given below characterize the communications element of a system by a series of *VOTER* processes, each concerned with validating the output of some task. They repeatedly obtain information (on a channel *task*) from the execution of a task; for simplicity we assume that it is clear from a boolean flag passed along this channel whether or not the execution succeeded[2]. Provided that at least two successful executions occur in each cycle, the voter will successfully agree on the output values of that task (signalled by the *pass* event) and wait for the end of a frame (indicated by the *sync* event).

The actual permuted schedules can be modeled in an abstract way by providing each task with a "source" of executions – we do not need to model the actual schedules explicitly, but only to capture the condition which a reasonable set of permutations will satisfy in the presence of transients: each task will be executed four times in each cycle, of which no more than one will be corrupted. This importance of this model is that in later documents we will be able to constrain these sources by placing them in parallel with particular schedulers, and verify that those schedulers do meet the following correctness condition: We can combine source processes for each of the tasks under consideration with the voters, and demonstrate that the voters are always satisfied that sufficient executions have succeeded. In terms of the model below we must show that each frame contains a *pass* event for all tasks.

timing.csp: A model supporting verification of permuted schedules and associated voting.

(c) Formal Systems Design & Development, Inc, 1994

Originated by: Richard Chapman / Michael Goldsmith This version:
```
-- $Id: timing.csp,v 2.0 1994/12/16 17:44:03 dave Del $
```

Basic type definitions:

```
TASK = { 0, 1, 2, 3, 4 }
BOOL = { true, false }
```
The set of task names
and validity values

Channel declarations:

The following channels indicate completion of a task instance, and pass a flag indicating its success or failure:
```
pragma channel task : TASK . BOOL
```

pass is used to indicate successful acquisition of sufficient correct copies of the output of a task by a voter

[2]This is simply an abstraction of the actual voting and comparison of data.

```
pragma channel pass : TASK
```

```
pragma channel sync
```
indicates the end of a frame, and

```
pragma channel work
```
is an interleaved event simply representing the occurence of some internal computation (typically related to comparing the results of different instances of a task).

The communication part of the system defines a voting process for each task in the system:

```
COMMS =                     ((((VOTER(0, 2)
        [| {| sync |} |] VOTER(1, 2))
        [| {| sync |} |] VOTER(2, 2))
        [| {| sync |} |] VOTER(3, 2))
        [| {| sync |} |] VOTER(4, 2))
```
The voters synchronize on the sync signal, ensuring that all tasks are validated with respect to the same cycle boundaries.

The voter process itself accepts inputs on task, and when sufficient valid instances have been counted, it outputs a pass signal recording the task number.

```
VOTER (i, n) =
        task.i ? ok ->
                if ok
                then        if n == 2
                        then VOTER (i, 1)
                        else if n == 1
                        then work -> pass ! i -> FRAME (i)
                        else work -> VOTER(i, n) never happens
else work -> VOTER (i, n)
```

After successful output, the voter waits for completion of the cycle. It is still prepared to accept (and discard) further completion signals.
```
FRAME (i) =
        (sync -> VOTER (i, 2)) [] task.i ? any -> work -> FRAME (i)
```

The tasks are represented by a combination of source processes, each enforcing the condition that sufficient correct instances of the appropraite task occur in each cycle. Their only interaction at present to to synchronize on the end-of-cycle signal.

Future models will exploit this framework by combining particular scheduler patterns in parallel with these processes.

```
SOURCES =             ((((SOURCE(0)
       [| {| sync |} |] SOURCE(1))
       [| {| sync |} |] SOURCE(2))
       [| {| sync |} |] SOURCE(3))
       [| {| sync |} |] SOURCE(4))
```

At the start of each cycle, four instances of the task are required, and no incorrect instances have been observed.

```
SOURCE (i) = NOTYETBROKEN (i, 4)
```

This process represents a source which has yet to observe an unsuccessful execution. It permits a synchronization signal and a return to its initial state if all four instances of the task have been observed. (n holds the number yet to be seen). If only one task remains to complete, it will allow the end-of-frame signal, assuming the last instance of the task to have failed. In other cases, it waits for a completion signal and decerments the counter if the execution succeed, or moves to the $ALREADYBROKEN$ state if it failed.

```
NOTYETBROKEN (i, n) =
        if n == 0
        then sync -> SOURCE (i)
        else if n == 1
        then sync -> SOURCE (i) [] task.i ? ok -> sync -> SOURCE (i)
        else task.i ? ok ->
                if ok
                then NOTYETBROKEN (i, n-1)
                else ALREADYBROKEN (i, n-1)
```

In each cycle, once a single erroneous execution has been observed, the remaining n must complete successfully.

```
ALREADYBROKEN (i, n) =
        if n == 0
        then sync -> SOURCE (i)
        else task.i ! true -> ALREADYBROKEN (i, n-1)
```

In the current version, our system model is simply the combination of the task execution part and the communication part.

```
SYSTEM = SOURCES [| {| sync, task |} |] COMMS
```

The model show above does have some practical disadvantages, however. The computation (represented by *work*), and the task completion (*task*) signals are arbitrarily interleaved, and the number of possible states whose behavior must be considered (either by automatic or manual analysis) grows very rapidly as the number of tasks considered increases. We can reduce this growth by noting that the voters would in practice differentiate between processing and communication, possibly refusing to exchange more data until the work associated with previous communication was complete. In the CSP model, we wish to distinguish the *work* events from the *task* communications by a difference in priority.

Encoding this distinction in a form suitable for use with the current **FDR** tool is quite difficult. We must consider the voters as constituting a single process which maintains a vector of information, holding a count of executions of each task in each element of the vector. This allows us to replace the interleaving of communications in the previous model with a sequential form which maintains the desired relationship between *task*, *work* and *pass* events. A model which uses the SML embedding techniques supported by **FDR** to implement this scheme is given in Appendix A.

A much more satisfactory model incorporating priorities to distinguish internal and external activity in a system or sub-system can be built on the basis of Dr Goldsmith's work described in another part of this report [5]. The **FDR** tool currently under development ([4]) will, when extended by the prioritization operator developed by this project and discussed in [5], allow such models to be written in the simpler style of the model given above, while maintaining the semantic distinctions and practical efficiency of that given in Appendix A. Such a framework will be essential for the extension of this framework into a tool for checking the transient-tolerance of a specified set of permutations of a schedule.

5.1.2 Permitted Permutations

For the use of permuted schedules to be valid, we must be able to find an appropriate number of viable schedules for the task set which makes up the application program. This potentially difficult task is subject to some non-obvious constraints, as we will show here. Consider the data dependence relation and four schedules in Figure 6. Assume all tasks take equal time to execute. The instance of task one in any given frame for processor four computes a value that will not be computed by the other three processors until the beginning of the next frame. Suppose that we are voting on task seven only, and that processor four's value for task seven in some frame turns out to be in error. That means that the value already computed for task one for the next cycle must also be invalidated. Somehow, processor four must at some future point "catch up" – compute values for tasks one through six, since they are not voted, and be able to compute a valid value for task one in advance of the other three processors if it is to resume executing its schedule. Processor four will never be able to do it, since to do so it must after some number of frames k have computed $7k+1$ values in $7k$ time slots.

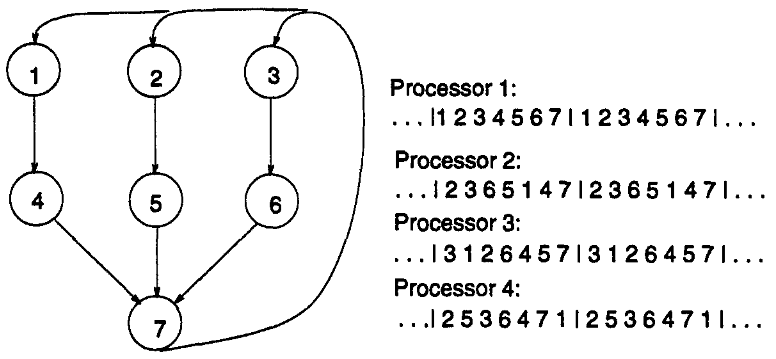

Figure 6: Disallowed set of permutations

Consequently we require permissible permuted schedules to obey the property that within a single frame the instances of a task on all replicated processors correspond to the same iteration. That is, if we let $O_k^p(t_i)$ represent the output from task t_i on processor p in frame k, then in the absence of failures,

$$\forall k. \forall p' \in \mathit{Processors}. O_k^{p'}(t_i) = O_k^p(t_i)$$

5.2 Voting in permuted schedules

The standard method of implementing systems whose state information is maintained by an interactive consistency algorithm such as Oral Messages is to arrange that applications use the agreed value of a state variable in place of the locally calculated one when a task requires that variable as input. In effect, we must arrange our voting and computation in such a way that a sufficient set of state values are always agreed by voting before they are used by tasks which depend on them.

These constraints further complicate the process of finding suitable schedules for a set of redundant processors which, as the previous section and reference [7] show, is already subject to significant constraints.

We therefore seek to relax this "vote before use" condition, in order to introduce sufficient flexibility to support permuted schedules and to gain other benefits:

- Relaxing the restriction places fewer constraints on the sequence in which tasks can be executed, potentially reducing any need for one processor to be idle while others compute values which need agreement.

- In systems where computation (by the application) and communication (which constitutes much of the voting process) use different resources, they can be overlapped to a greater extent if the strict ordering is relaxed, leading to significant performance benefits.

Rather than requiring the replicated processors to vote on the outputs of all tasks, we only specify voting on outputs to actuators and on a set of tasks satisfying a minimal voting condition ([9], p. 60), which we call a *basis set* of tasks. If permuted schedules were not permitted, voting could occur immediately upon completion of the task to be voted (by all the processors) which should happen simultaneously, given the requirement that we know absolute execution times for all tasks ([1], p. 1).

However, if permuted schedules are permitted, voting must be delayed at least until a plurality of processors have computed some output value for the task to be voted. The point within a frame at which a given basis task's output can be voted is statically determinable and thus the communication events necessary to carry out the voting can be incorporated into the schedule.

A consequence of the necessary delay in voting is that it becomes likely that a processor that is the first to run some basis task will have to use its locally computed, not yet voted, value for the output of that basis task until the voted value becomes available. If the voted value agrees with the locally computed value, all is well, but if the locally computed value is invalidated by the vote, the processor must begin recovery of a number of tasks. The results of not only the voted task but also all other comparable tasks (either as ancestors or descendants) in the transitive closure of the data dependence relation between tasks [6]become invalid, as in the example of Figure 7. Upon invalidation of any tasks, there must exist some sequence of actions that the recovered processor can take to restore all invalidated tasks to having valid input data at the appropriate times in each frame, according to its schedule.

We can shorten the waiting period required for voting by not requiring a task to wait for results from all four replicated processors. Two values in agreement are enough evidence for a processor to conclude that it has the voted value, so why wait for all four? However, a processor that proceeds before receiving input from all processors contributing to a vote must ensure that those messages it plans to ignore are properly dealt with if they should arrive at some future point. We propose that a processor deciding to ignore communications from some other processors should spawn a *sacrificial buffer* process that will catch those messages when they do arrive, or notify the processor if they never do (that is, if the buffer process receives another request from its own processor to wait on a value from the peer processor before it has received a first value from the peer). This fact is evidence of a failure either in

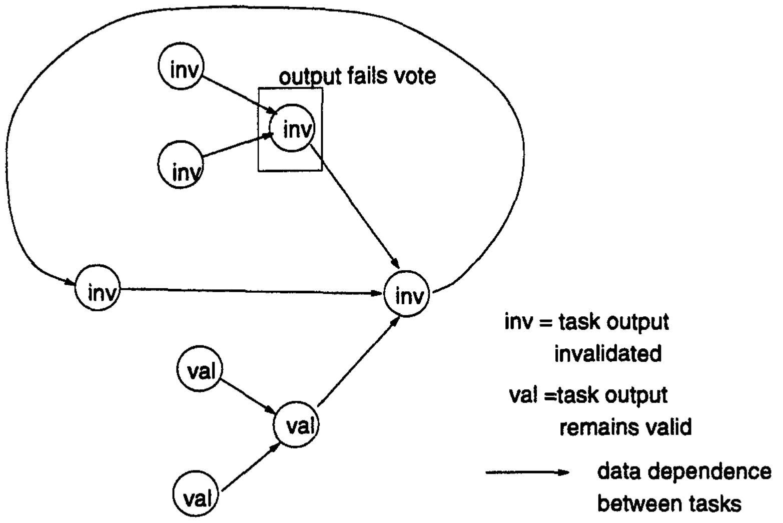
inv
output fails vote
inv
inv
inv
inv
val
val
val
inv = task output invalidated
val =task output remains valid
data dependence between tasks

Figure 7: Tasks that must be invalidated

the peer processor or the link. We outline a specification for such a buffer and the code that the processor spawning the buffer must run in the next section of this note.

There are obviously a number of potential difficulties which relaxing the voting pattern in this way may introduce. It is obviously vital that the tolerance of Byzantine faults should not be reduced, and indeed this fact does follow from the properties of the network shown in Section 3. Because we do eventually obtain as much information on the correctness of a value as is available in the straight-forward implementation of the OM algorithm our ability to detect and correct errors is unaltered, although detection of an error may be slightly delayed when compared with a fully sequential voting arrangement.

The most significant penalty incurred by the change is that transient errors in the data held by a processor are no longer corrected "automatically": if state data is always agreed with a processor's peers before being used then a single corrupt value will not be passed to any instance of the tasks which use it, and if the fault causing corruption is transient it will be corrected when the value is next modified. This is obviously not the case if a processor continues using the corrupt value without checking it. The process of recovery from transient errors will be considered further in a later section.

6 Distributed Models of a Voter

Below we develop a model of a voting mechanism that can be used when processors are running permuted schedules. Rather than requiring the communications processor to spawn a process to catch "late" data values transmitted by peer processors, we add four local processes, running concurrently with the voting mechanism, which we call *smart buffers*. Each smart buffer is responsible for reception of messages from one of the peer processors, for maintaining the local processors' decision about the state of that peer processor (good, faulty, or dead), and for conveying information about recent communications with the peer processor when requested.

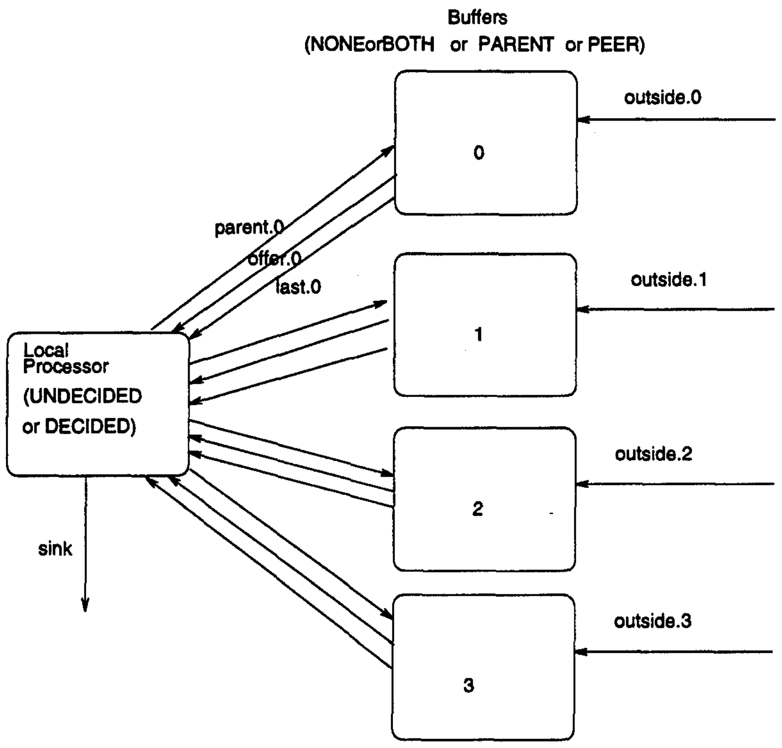

Figure 8: Local processor and buffers for communication with peers

The buffer has three major states. The current state is determined by which of the local (or parent) processor or the remote (or peer) processor it has last communicated with. The communications processor has two major states. We say it is *decided* if it has determined the valid value for its task for this frame as a result of comparisons between values sent by the remote processors for the task's value this frame. Otherwise it is *undecided* .

sb.csp: Distributed model of communication and voting

(c) Formal Systems Design & Development, Inc, 1994

Originated by: Richard Chapman This version:

```
-- $Id: sb.csp,v 2.0 1994/12/16 17:44:03 dave Del $
```

There are four processors, numbered 0 to 3.

```
FOUR = {0, 1, 2, 3}
```

The value computed by a task may be one of two valid values, or a mesage from a processor to ignore its value

```
DATAVAL = { one, zero , ignoreme }
```

A processor may assign one of its peers any of the following status values

```
STATUSVAL = { bad, ok, dead }
```

The system present at each processor consists of the process running on the processor itself, plus four concurrent processes representing buffers to receive values from the remote processors (we could handle the value computed locally as a special case, but do not). The 4 buffer processes send data values to the local processor over the offer channels

```
pragma channel offer : FOUR . DATAVAL
```

The local processor can communicate its voted value for the task to the buffers over the parent channels

```
pragma channel parent : FOUR . DATAVAL
```

Channel last is used by a buffer to communicate the status of the peer with which it communicates to the local processor.

```
pragma channel last : FOUR . STATUSVAL
```

Sink is the channel on which the processor broadcasts a valid value for a task to the outside world, once per frame

```
pragma channel sink : DATAVAL
```

Synchronization signal sent between successive iterations of a task:

```
pragma channel frame
```

channels for peer processors to send values to sacrificial buffers:

```
pragma channel outside : FOUR . DATAVAL
```

Code for the local processor to interface with the sacrificial buffer

A processor determines the valid value for a task by receiving the values computed by its peer processors and comparing that value with its own locally-computed value for that task. Once a processor has received the same value from two processors (one of which could be itself), it can conclude that the value it received more than once is the valid value.

At any time, the communications processor will be running one of two processes (UNDECIDED or DECIDED) for each task.

The process UNDECIDED represents the state of the communications processor when it has not yet received two values in agreement for some task. The set I is the set of other processors from which the processor has yet to receive a value, and the set A is a set of ordered pairs (processor, value) that have been received. If a peer processor sends an ignoreme message, its number is removed from I. If a peer processor sends any other data value, that value is compared to previously received values. If that value is found in the list, the processor concludes that it has the valid value and behaves like process DECIDED (keeping track in its first parameter of which peer processes from which it has not yet heard), else it adds the value to the set of received values and behaves like UNDECIDED

```
UNDECIDED (I, A, untimed) =
        ([] i: I @ offer.i ? x ->

        if (x == ignoreme) then
                UNDECIDED (diff (I, {i}), A, untimed) else
        if (member (x, { head (tail (x1)) | x1 <- A })) then
                DECIDED (diff (I, {i}), x, untimed)
        else
                UNDECIDED(diff(I,{i}), union(A,{ <i,x> }), untimed)
        )
```

Once a process has decided the valid value for a task (parameter x in the process DECIDED, below) , it can use that value for further computation, but must rely on a process (a concurrently running "smart buffer") to handle reception of the remaining *transmissions of values for that task by the other peer processors (whose numbers are in set I). The smart buffer must also be responsible for notifying the local processor if any peer processors fail to respond*

As soon as the local processor decides the valid value, it sends value over channel parent to the buffers, who will use it in determining the status (good, bad, or dead) of the peer processors

When the local processor has finished notifying the buffers, it announces the value it determined to be valid to the outside world over channel sink, then waits for the frame synchronization event and starts over.

```
DECIDED (I, x, untimed) =
        ([] i:I @ parent.i ! x -> last.i ? s ->
                 DECIDED (diff (I, {i}), x, untimed))
        []
        (if (empty (I)) then
                sink ! x ->
                        if untimed
                        then frame -> UNDECIDED (FOUR, {}, untimed)
                        else UNDECIDED (FOUR, {}, untimed)
        else STOP)
```

Code for a smart buffer running in parallel with a processor

The smart buffer has one of several states depending on whom it heard from last: the PARENT (local) processor, the PEER (remote or local) processor, or NONEorBOTH (ready to receive a message from either).

Initially, a smart buffer has not heard either from its parent (via channel parent) or from any peer processor (via channel outside). It is ready to communicate via either channel, and change its state based on which it hears from first

```
NONEorBOTH (i, s, untimed) =
        (parent.i ? x -> last.i ! s -> PARENT (i, x, untimed))
        []
        (outside.i ? y -> PEER (i, y, s, untimed))
```

A smart buffer that has last heard from its parent knows the value the parent decided was valid (x), and is waiting to hear that value from the PEER processor. If it does *hear a value from the peer, it computes a statusval for the peer (okay or bad, depending if the value sent by the peer is the same as that decided upon by the parent) and resumes listening for either the parent or the peer*

If the frame event occurs before the buffer hears from the peer, it assumes the peer is dead and changes its status accordingly. If frame events are not being used, if the buffer hears from the parent again before hearing from the peer, it sends a message to the parent (over channel last) indicating that it believes the peer is dead

```
PARENT (i, x, untimed) =
        (outside.i ? y ->
                if untimed
                then frame ->
                  NONEorBOTH (i, if x==y then ok else bad, untimed)
                else
                  NONEorBOTH (i, if x==y then ok else bad, untimed))
        []
        if untimed
        then frame -> NONEorBOTH (i, dead, untimed)
        else parent.i ? xx -> last.i ! dead -> PARENT(i, xx, untimed)
```

A buffer that has heard from the peer processor sends the value it heard to the parent via the offer channel. After sending an offer it waits for the frame synchronization event and then resumes waiting to hear from either the parent or the peer

If the processor receives a value from the parent before it can offer the value from the peer to the parent, obviously the parent already had enough values from other buffers to make a decision, so the buffer sends the status value from the last frame to the processor and then computes a new status value for this frame, arrived at by comparing the value received from the parent this frame with the value received from the peer this frame, then waits for the frame synchronization event, and then listens for either the parent or peer at the start of the next frame

```
PEER (i, y, s, untimed) =
        (offer.i ! y ->
                if untimed
                then frame -> NONEorBOTH (i, ok, untimed)
                else NONEorBOTH (i, ok, untimed))
        []
        (parent.i ? x -> last.i ! s ->
                if untimed
```

```
                        then frame ->
                          NONEorBOTH (i, if x==y then ok else bad, untimed)
                        else
                          NONEorBOTH (i, if x==y then ok else bad, untimed))
```

The system consists of a processor (initially running UNDECIDED(FOUR,,true) and four smart buffers, one for each of the four peer processors. We could optimize this to three and handle the locally computed value entirely within the local processor if desired

```
uSYSTEM = UNDECIDED (FOUR, {}, true)
          [| {| last, offer, parent, frame |} |]
          ((((                      NONEorBOTH (0, ok, true))
            [| {frame} |] NONEorBOTH (1, ok, true))
            [| {frame} |] NONEorBOTH (2, ok, true))
            [| {frame} |] NONBEorBOTH (3, ok, true))
```

When we hide the communication between the four buffers and the local processor we get:

```
UntimedSystem = uSYSTEM \ {| last, offer, parent |}
```

If we dispense with the frame synchronization events:

```
tSYSTEM = UNDECIDED (FOUR, {}, false)
          [| {| last, offer, parent |} |]
          ((    NONEorBOTH(0,ok,false) ||| NONEorBOTH(1,ok,false)
            ||| NONEorBOTH(2,ok,false) ||| NONEorBOTH(3,ok,false))
           [| {| frame |} |] frame -> ZERO)

SystemWithoutTiming = tSYSTEM \ {| last, offer, parent |}
```

In order to assert that frame never occurs, the process above includes a transition to ZERO if frame should ever occur. Because ZERO is the "worst-possible" process in the Failures-Divergence model, this will result in tSYSTEM failing any non-trivial refinement check, should frame occur.

```
ZERO = ZERO |~| ZERO
```

The bottom process is represented as a non-deterministic choice for purely technical *reasons. (***FDR***cannot itself successfully compile the more usual definition* `-- ZERO = ZERO`*).*

The specification for the System described above

MAJORITY's three parameters are sets of processes. I represents the processors which have not contributed a value for the task this frame, while Zeroes and Ones are respectively, the sets of processors that have contributed a value of zero and a value of one this frame (note that a processor sending an ignoreme message drops out of I without ever appearing in Ones or Zeroes that frame)

If the size of Zeroes exceeds one, the specification may output a value of one on channel sink. It may output a zero on sink if the size of Zeroes exceeds one. If both sets exceed one in size, the specification produces a nondeterministic result

If the set I is not empty at the time the specification sends its decision on the valid value on channel sink, the specification must behave like process CHOMP until the end of the frame

CHOMP acts as a buffer to receive any values transmitted by the peer processors after the local processor has produced output during that frame on channel sink (in the system, the smart buffers handle this)

```
MAJORITY (I, Zeroes, Ones) =
        (outside ? i:I ? x ->
                if (x == zero)
        then MAJORITY (diff (I, {i}), union (Zeroes, {i}), Ones)
                else if (x == one)
                then MAJORITY(diff(I, {i}), Zeroes, union(Ones, {i}))
                else MAJORITY(diff(I, {i}), Zeroes, Ones))
        []
        ((if 1 < card (Ones)
          then sink ! one -> CHOMP (I)
          else STOP)
         |~|
          if 1 < card (Zeroes)
          then sink ! zero -> CHOMP (I)
          else STOP)
        []
        if empty (I)
        then (if card (Ones) < card (Zeroes)
                then sink ! zero -> CHOMP (I)
                else if card (Zeroes) < card (Ones)
                then sink ! one -> CHOMP (I)
                else (sink ! zero -> CHOMP (I) |~|
                      sink ! one -> CHOMP (I)))
        else STOP

CHOMP(I) =
        (outside ? i:I ? x -> CHOMP (diff (I, {i})))
        []
        frame -> MAJORITY (FOUR, {}, {})
```

Initially, no processors have sent values of one or zero, and the specification is waiting on a value from all four processors

```
SPEC = MAJORITY (FOUR, {}, {})
```

Here we model the data distribution phase. A value on channel source is copied to each of the four outside channels by processes INJECTOR and SPREAD. A frame synchronization event is expected between sucessive inputs on channel source (and consequently between any successive pair of outside.i events).

```
pragma channel source : DATAVAL

 INJECTOR = source ? x -> SPREAD (FOUR, x)

SPREAD (I, x) =
        if empty (I)
        then frame -> INJECTOR
        else |~| i:I @ outside.i ! x -> SPREAD (diff(I, {i}), x)

FRAMESYNCH =
        (INJECTOR [| {| outside, frame |} |] UntimedSystem) \
        {| outside, frame |}
```

We introduce the possibility of a fault on channel outside.0 The same symmetry arguments previously made apply here.

```
XFRAMESYNCH =
        ((INJECTOR \ {| outside.0 |} ||| CHAOS ({| outside.0 |}))
        [| {| outside, frame |} |]
        UntimedSystem) \ {| outside, frame |}

CYCLICINPUT =
        (INJECTOR [| {| outside |} |] SystemWithoutTiming) \
        {| outside, frame |}

XCYCLICINPUT =
        ((INJECTOR \ {| outside.0 |} ||| CHAOS ({| outside.0 |}))
        [| {| outside |} |]
        SystemWithoutTiming) \ {| outside, frame |}

pragma channel forward : FOUR . DATAVAL
```

A 1-place buffer is introduced along each of the four outside.i channels, fed by the INJECTOR

```
BOUNDEDDELAY1 =
        ((INJECTOR[[outside<-forward]]
         [| {| forward |} |]
         (   BBUFFc(1,forward.0,outside.0)
         ||| BBUFFc(1,forward.1,outside.1)
         ||| BBUFFc(1,forward.2,outside.2)
         ||| BBUFFc(1,forward.3,outside.3))
        ) \ {| forward, frame |}
        [| {| outside |} |]
        SystemWithoutTiming) \ {| outside |}
```

The possibility of a fault is allowed on channel outside.0

```
XBOUNDEDDELAY1 =
        (((INJECTOR \ {| outside.0 |} ||| CHAOS ({| outside.0 |}))
         [| {| forward |} |]
         (   BBUFFc(1,forward.1,outside.1)
         ||| BBUFFc(1,forward.2,outside.2)
         ||| BBUFFc(1,forward.3,outside.3))
        ) \ {| forward, frame |}
        [| {| outside |} |]
        SystemWithoutTiming) \ {| outside |}

pragma channel tock
```

REGULATOR runs in parallel with the distribution of values to the outside channels to guarantee that all transmissions of values on the the outside.i channels (i is an element of parameter I) occur within N tocks, and that following the completion of one frame of events on the outside channels, M tocks elapse before the outside events for the next frame commence

```
REGULATOR (I, M, N) =
        tock -> REGULATOR (I, M, N) []
        outside ? i:I ? x -> REGULATOR' (I, M, N, diff (I, {i}), 0)
```

J is the subset of I whose outside channels have had no event this frame yet, while n is the number of tocks elapsed since beginning of this frame's outside events

```
REGULATOR' (I, M, N, J, n) =
        (if n < N
         then tock -> REGULATOR' (I, M, N, J, n+1)
         else STOP)
        []
        (outside ? i:I ? x -> REGULATOR' (I, M, N, diff (J, {i}), n))
        []
        (if empty (J)
         then DELAY (M); REGULATOR (I, M, N)
         else outside ? i:I ? x -> REGULATOR'(I,M,N,diff(J,{i}),n))
```

DELAY ensures that n tocks elapse between last outside event of one frame and first outside event of the next frame

```
DELAY (n) = if 0 < n then tock -> DELAY (n-1) else SKIP
```

Adding the REGULATOR to the rest of the already-developed system gives:

```
REGULATED1 =
        (((INJECTOR[[outside<-forward]]
          [| {| forward |} |]
          (   BBUFFc(1,forward.0,outside.0)
          ||| BBUFFc(1,forward.1,outside.1)
          ||| BBUFFc(1,forward.2,outside.2)
          ||| BBUFFc(1,forward.3,outside.3))
          ) \ {| forward, frame |}
         [| {| outside |} |]
         REGULATOR (FOUR, 2, 2)) \ {tock}
```

```
          [| {| outside |} |]
          SystemWithoutTiming) \ {| outside |}
```

and if we allow channel outside.0 to be faulty:

```
XBOUNDEDDELAY1 =
          (((INJECTOR \ {| outside.0 |} ||| CHAOS ({| outside.0 |}))
           [| {| forward |} |]
           (    BBUFFc(1,forward.1,outside.1)
           ||| BBUFFc(1,forward.2,outside.2)
           ||| BBUFFc(1,forward.3,outside.3))
          ) \ {| forward, frame |}
          [| {| outside |} |]
          SystemWithoutTiming) \ {| outside |}
```

Suppose we want to add the restriction that an event occurs on each of the four outside channels every k tocks, where k can vary from frame to frame, but must always be within some bounds of N tocks. Say, for some other integers A and B, that k can never be less than N - A nor more than N + B . Process PWB guarantees that rate of events on channel c:

```
PWB (c, N, B, A) = PWB' (c, N, B, A, 0)

PWB' (c, N, B, A, n) =
          if n < N - B
          then tock -> PWB' (c, N, B, A, n+1)
          else if n < N + A
          then     (tock -> PWB' (c, N, B, A, n+1))
                |~| (c ? x -> PWB' (c, N, B, A, n-N))
          else c ? x -> PWB' (c, N, B, A, n-N)

PWBs =             (((PWB (outside.0, 5, 2, 2)
          [| {tock} |] PWB (outside.1, 5, 2, 2))
          [| {tock} |] PWB (outside.2, 5, 2, 2))
          [| {tock} |] PWB (outside.3, 5, 2, 2))
```

Adding the restrictions on the buffers to the system yields:

```
PWB1 =  ((INJECTOR[[outside<-forward]]
           [| {| forward |} |]
           (   BBUFFc (1,forward.0,outside.0)
        ||| BBUFFc (1,forward.1,outside.1)
        ||| BBUFFc (1,forward.2,outside.2)
        ||| BBUFFc (1,forward.3,outside.3))
      ) \ {| forward, frame |}
     [| {| outside |} |]
     (PWBs \ {tock}
      [| {| outside |} |]
      SystemWithoutTiming)) \ {| outside |}

RUN(X) = [] a:X @ a -> RUN(X)
```

But we must also model the possibility that our faulty channel (channel outside.0) does not produce its values in a timely fashion:

```
XPWBs =            ((PWB (outside.1, 5, 2, 2)
        [| {tock} |] PWB (outside.2, 5, 2, 2))
        [| {tock} |] PWB (outside.3, 5, 2, 2))
```

The system becomes:

```
XPWB1 = ((INJECTOR[[outside<-forward]]
          [| {| forward |} |]
          (   RUN ({|forward.0|}) ||| CHAOS ({|outside.0|})
          ||| BBUFFc (1,forward.1,outside.1)
          ||| BBUFFc (1,forward.2,outside.2)
          ||| BBUFFc (1,forward.3,outside.3))
         ) \ {| forward, frame |}
        [| {| outside |} |]
        (XPWBs \ {tock}
         [| {| outside |} |]
         SystemWithoutTiming)) \ {| outside |}
```

Generic 1-place buffers

An N-place buffer receiving input on channel source and producing output on channel sink:

```
BBUFF (N) = BBUFFc (N, source, sink)
```

BBUFFc is an N-place buffer also taking the names of its input and output channels as parameters:

```
BBUFFc (N, in, out) = BBUFF' (N, 0, <>, in, out)
```

There are two more components to the state of BBUFF': it's current contents (the list of values s) and the number of items currently stored in the buffer:

```
BBUFF' (N, n, s, in, out) =
        if n == 0
        then in ? x -> BBUFF' (N, 1, <x>, in, out)
        else if n == N
        then out ! head (s) -> BBUFF' (N, n-1, tail (s), in, out)
        else
          ((out ! head (s) -> BBUFF' (N, n-1, tail (s), in, out))
        [] ( in ? x -> BBUFF' (N, n+1, s^<x>, in, out)
            |~| out ! head (s) -> BBUFF'(N,n-1,tail(s),in,out)))
```

7 Recovery from transient errors

In order to arrange that a process can recover from a transient error even if values are not always agreed before use, we must arrange that:

- An FCR which has suffered a transient fault can detect the resulting error and thus take appropriate recovery action.
- While such a node is recovering the erroneous values, it does not promote failure in the other FCRs in the system.
- During recovery, all significant state values will be recovered from correctly functioning peer nodes, and sufficient computation will be performed to maintain and re-generate state which is not directly communicated.

The first two requirements are relatively undemanding; the first is effectively a constraint on the types of errors that we can expect to tolerate. One implication which must be considered, however, is that the design will have to distinguish between a "local" value and a value received from a peer when performing comparisons[3]. If the "local" value disagrees with the majority, then a node should be considered to have suffered a fault and should attempt to recover the relevant values. The second requirement is also trivial for some classes of faults. If no further errors occur in the

[3]Note that the models in Section 3 did not need to make this distinction.

portion of the system state related to that which we are attempting to recover (using the dependency relationship discussed in Section 5.2), we are guaranteed that three correct values will always be available to non-faulty nodes, and thus any erroneous output from a processor in the process of recovering will be ignored. We can gain some advantage, however, from allowing a processor which has detected a transient fault to notify its peers of this fact: the three remaining FCRs may then be able to survive a second non-independent error by moving to a 2-out-of-3 voting scheme.

The last requirement in the above list is the most difficult to satisfy. If, for reasons of timing drift or because of a permuted schedule, a processor suffering a transient fault was the earliest node to compute the relevant values in each frame, we may never be able to guarantee that it can obtain a timely, reliable value from its peers. We can suggest several approaches to this problem, including

- Arranging that the entire system reverts to a fall-back schedule which does guarantee to agree all state values by voting.
- Finding (where possible) a "reversionary" schedule for the failed processor alone.
- Arranging that the recovering processor and one of the fault-free peers change to pair of reversionary schedules which transfer corrected data to the recovering node while maintaining just sufficient of the normal behavior to ensure correct system operation.

These possibilities are discussed below:

7.0.1 A fully-voted reversionary schedule

We might propose the following scheme of operation: whenever it is determined that a processor has computed an invalid result for some task, that processor will be asked to sit idle until the end of the current frame, at which time all processors will stop running their particular permuted schedules and start running a single already-agreed-upon schedule. Thus, during the next frame, all processors will be running the same schedule. We know because of our requirement from Section 5.1.2 that only permitted permutations were used, that the output records for all non-invalidated replicated instances of all tasks contain the same data, and that if the invalid processor has not failed, it can recover valid values for all tasks (values derived from voted values of all tasks in the basis set) by the end of the next frame. If no notifications of invalid results are received by any processors during the execution of the recovery frame, the processors switch to their particular permuted schedules at the end of that recovery frame.

This scheme does, however, limit many of the advantages which motivate our use a relaxed voting scheme. If we must be able to operate on a fully-voted schedule, we cannot take advantage of the performance benefits which overlapping communication and computation will bring. In particular, there will be many sets of permuted schedules for which no suitable fixed schedule with complete voting will exist. Further, at least for the duration of the recovery, we will have lost the benefits of permuted scheduling.

7.0.2 A single reversionary schedule

This alternative again suffers from the disadvantage that for many applications there will be no single order of task execution which requires data only after it has been made available by one of the peer nodes. If only the recovering processor reverts to this schedule, however, we can remove some of the constraints limiting our execution order. In particular, the reversionary schedule need not calculate any outputs or other values which are calculated afresh in each cycle; it need only perform that minimum computation which is necessary to maintain the relevant node state. In terms of the data dependency graph, we need only execute those tasks which occur on the cycles through the initial erroneous task. Branches which do not form part of a cycle may be neglected, and indeed as we saw above, there are concrete advantages to be gained from a processor informing its peers that it should be ignored in any votes which take place during its recovery.

Due to the difficulty of finding a suitable schedule (if one exists), this technique will obviously be limited in its application, particularly as to exploit the potential benefits of permuted scheduling, we must find a number of recovery schedules, each capable of re-generating a particular set of corrupt values while maintaining the outputs and correct behavior of the uncorrupted elements of the application.

7.0.3 Partial reversionary schedules

In an attempt to avoid some of the difficulties associated with both of the above schemes, we propose considering a method which combines some features from each. Finding a single processor schedule which is compatible with the permuted schedules already running on fault-free processors, as required by the previous scheme is clearly more difficult than the problem of finding two schedules which suffice to transfer some part of the system schedule to the recovering processor. This latter problem is simplified further if we allow the fault-free partner in such a recovery to neglect some of its output calculations (on the basis that there will still be duplicate correct values generated by the remaining pair) – we clearly lose tolerance to further faults during this operation, but the practical value of such resilience to two faults will obviously depend on the reliability analysis of a particular application. We do require a mechanism for identifying which processor should assist in the recovery when a fault is detected, but even here we may have a degree of choice over which of the remaining permuted schedules is most suitable for correcting the specific error identified.

The greatest cost of this approach is the effort of identifying sufficiently many reversionary schedules to maintain the benefits of temporal redundancy. We should note, however, that this is a task which is determined entirely by the static schedules chosen, and thus need not be carried out in a time-critical environment. The run-time penalty should be little more than identifying which regions of the data-dependency graph have been invalidated and looking up the appropriate recovery schedules in a pre-computed table.

To ease this task, it is perhaps desirable to consider the data-dependency graph as being divided into *software containment regions* which are treated as either being believed correct or believed corrupted as a whole. These regions must obviously contain the transitive closure of the relevant voted state variables, as discussed in Section 5.2. We also note that the fault-free processors initiating a recovery must agree on the identity of the FCR to be recovered and on the particular peer who will enter the assisting reversionary schedule. This information is, however, amenable to voting in a similar manner to other values, and is only required when votes are taken on state data – it need not apply to the agreement of output values, for example.

8 Conclusions

We do not expect this document to be viewed as a complete analysis of the FTP design, but to be seen as a working paper describing the state of various threads of analysis and modeling. One of the primary purposes of this paper is indeed, to present some ideas for comment from Draper representatives who, we hope, will be able to view them in the context of their greater familiarity with the concerns of the application domain. Significant features of recent developments include:

- Clarification of arguments based on symmetry which can be used to establish properties of the full FTP system from properties of a single voter. This work is sufficiently established that we feel a formal mathematical proof of the approach could be given. It is a result which will be particularly important in the future development of models which include more detail about the operating mechanisms of their components. It has already assisted in the rest of this work.

- Moving toward a less abstract model bearing a closer resemblance to the implementation, we have gained significant understanding of the problems faced in several key areas. These include
  - tolerance of transient faults,
  - recovery after transient errors, and
  - the benefits to be gained from temporal redundancy and permuted scheduling.

- We have presented models of the FTP consistency algorithm which include explicit timing information in both synchronous and semi-asynchronous models. These models include sufficient information about the communication mechanism to investigate the need for non-blocking and sacrificial buffers. We feel that these models approach the "Synchronous replicated" and "Asynchronous distributed" views of [1], although they still involve significant abstraction from the way in which the processing and voting elements operate, and the issue of establishing co-ordinated global timing has still to be addressed in detail.

The modeling which we have completed in this area is still highly abstract, but it provides important framework elements, and highlights those areas which place additional emphasis on new theories and tools.

The major prospects for future work on the demonstrator application lie in the following areas

- Our models are still very abstract in some areas: our models of communication are relatively close to transputer style implementations, but areas such as timing, clock synchronization and the mechanisms connecting hardware and software could benefit from more detail. Additional information may well allow us to relax some of our design constraints: for example our asynchronous timing model requires large margins in the specification of time-outs and cycle lengths, whereas slight improvements to the design we are formalizing may allow these margins to be reduced.
- At the implementation level, more detailed models of the interaction between software tasks is required, both in terms of specifying application timing constraints and especially in the relationship between communications hardware and software.
- The interface between communication, voting, and application software schedules is perhaps in greatest need of further formalization. Both this area and more general timing and scheduling issues will require the ability to model and distinguish systems using multi-processing on a single CPU and communications hardware supporting a single processor, as well as the theoretically simpler case of true multi-processor systems.

These prospects highlight some points of importance in the tool-development part of the project, in particular in the area of prioritization (as noted in Section 5.1.1) and possibly in assisting the modeling the interaction between varied hardware and software environments.

References

[1] N.A. Brock. Real-Time Scheduler: Natural Language Problem Statement. Technical report, Charles Stark Draper Laboratory, Inc., 1994. Deliverable D2.1 of SBIR N00014-93-C-0213, in [?].

[2] Neil A. Brock and Sharon L. Donald. Discussion of Errors and Their Effects on a HRT Scheduler. Technical report, The Charles Stark Draper Laboratory, Inc., 1994. Deliverable to SBIR N00014-93-C-0213, in [6].

[3] Formal Systems (Europe) Ltd, 3 Alfred St, Oxford OX1 4EH, UK. *Failures-Divergences Refinement (***FDR***), User Manual and Tutorial*, 1994. Contact D.M. Jackson; Tel: [+44] (0)1865 728460, Fax [+44] (0)1865 201114, E-mail: `dave@fsel.com`.

[4] P.H.B. Gardiner and M.H. Goldsmith. Inside **FDR 2**. Technical report, Formal Systems Design & Development, Inc., 1994. Adjunct to D1.2 of SBIR N00014-93-C-0213, in [?].

[5] M.H. Goldsmith. A CSP Priority Operator for **FDR 2**; Prototype Software for Discrete Real-time Extensions to **FDR**. Technical report, Formal Systems Design & Development, Inc., 1994. Deliverable to SBIR N00014-93-C-0213, in [?].

[6] M.H. Goldsmith et al. *N00014-93-C-0213: Third Quarterly Report.* Technical report, Formal Systems Design and Development, Inc., P.O. Box 3004, Auburn, AL 36831-3004, 1994.

[7] David M. Jackson and M.H. Goldsmith. Specifying Task Management; Single Processor Systems. Technical report, Formal Systems Design & Development, Inc., 1994. Deliverables D 2.2 and D 2.3 of SBIR N00014-93-C-0213, in [6].

[8] Patrick Lincoln and John Rushby. A Formally Verified Algorithm for Interactive Consistency Under a Hybrid Fault Model. In *Proceedings of 23rd Fault-Tolerant Computing Symposium*, 1993.

[9] B.L. Di Vito, R.W. Butler, and J.L. Caldwell. Formal Design and Verification of a Reliable Computing Platform For Real-Time Control – Phase 1 Results. Technical Memorandum 102716, NASA, 1990.

A Vector-based Model for Permuted Scheduling

abstract.csp: An FDR-1 model of a voter for permuted schedules

Originated by: Michael Goldsmith This version:
```
-- $Id: abstract.csp,v 2.0 1994/12/16 17:44:03 dave Del $
```

Define two vector operators in Standard ML: getnth returns the selected component of a sequence, setnth sets the selected component of the sequence to be the value specified.

Declare the function names as non-CSP definitions
```
pragma opaque "ML" getnth
pragma opaque "ML" setnth
```

Include the ML source code for the function implementations
```
pragma inline "ML" local
pragma inline "ML"   fun MLgetnth (0, a::x) = a
pragma inline "ML"     | MLgetnth (n, _::x) = MLgetnth (n-1, x)
pragma inline "ML"     | MLgetnth _ = raise SemanticError
pragma inline "ML"                      ("getnth: index too large")
pragma inline "ML" in
pragma inline "ML"   fun CSPgetnth [n, s] =
pragma inline "ML"     let val MLs = CheckSeq s
pragma inline "ML"         val MLn = NumberOf (CheckAtom n)
pragma inline "ML"     in MLgetnth (MLn, MLs)
pragma inline "ML"     end
pragma inline "ML"     | CSPgetnth x = raise TypeError
pragma inline "ML"       ("getnth: expected <number,sequence>,"
pragma inline "ML"       ^ " found "
pragma inline "ML"       ^ print_expression (EXPseqcomp (x, [])))
pragma inline "ML" end;
```

The following definition includes a call to print merely as development aid.
```
pragma inline "ML" local
pragma inline "ML"   fun revonto (a::x, y) = revonto (x, a::y)
pragma inline "ML"       | revonto (_, y)    =
pragma inline "ML"         (print "\nSTATE ";
pragma inline "ML"           map(print o print_expression)y;
pragma inline "ML"           print "\n"; y)
pragma inline "ML"   fun MLsetnth (0, _::x, v, y) = revonto (y, v::x)
pragma inline "ML"     | MLsetnth (n, a::x, v, y) =
pragma inline "ML"          MLsetnth (n-1, x, v, a::y)
pragma inline "ML"      | MLsetnth _ = raise SemanticError
pragma inline "ML"                    ("setnth: index too large")
pragma inline "ML" in
pragma inline "ML"   fun CSPsetnth [n, s, v] =
pragma inline "ML"      let val MLs = CheckSeq s
pragma inline "ML"          val MLn = NumberOf (CheckAtom n)
pragma inline "ML"          val _ = NumberOf (CheckAtom v)
pragma inline "ML"      in EXPseqcomp (MLsetnth (MLn, MLs, v, []), [])
pragma inline "ML"      end
pragma inline "ML"      | CSPsetnth x = raise TypeError
pragma inline "ML"        ("setnth: expected number,sequence,number>,"
pragma inline "ML"        ^ " found "
pragma inline "ML"        ^ print_expression (EXPseqcomp (x, [])));
pragma inline "ML" end;
```

Declare the relationship between the CSP names and the ML functions

```
pragma inline "ML" DefineMLFunction "getnth" CSPgetnth;
pragma inline "ML" DefineMLFunction "setnth" CSPsetnth;
```

The following sets and channels are equivalent to those in timing.csp

```
TASKS = { 0, 1, 2, 3, 4 }
BOOL = { true, false }
pragma channel task : TASKS . BOOL
pragma channel pass : TASKS
pragma channel work, sync
```

The communication model is now a single process with a vector argument

```
COMMS = JUDGE (<2,2,2,2,2>)
```

Initially this process accepts a termination signal, and if it is valid, moves to the DSZ state to decrement the appropriate value. If the execution was invalid, it performs the associated work (actually an abstraction of the decision process), and remains ready to accept another termination. Note that inputs are not accepted while work is being offered: this captures the prioritization of the "internal" action work over external communication

```
JUDGE (s) =
        task ? i ? b ->
```

```
                    if b
                    then DSZ (s, i, getnth (i, s))
                    else work -> JUDGE (s)
```

This process examines the count relating to task i and performs appropraite action. If the recent termination we the first successful one, the counter is decremented (without doing any work for the comparison), and the JUDGE returns to its initial state. If one previous successful execution had preceded this one, a comparison is performed, and the successful acquisition of good data is signalled on pass. Further successful executions are ignored (after the comparison, which is necessary to detect the occurence of a transient error, although not to determine the actual value required).

```
DSZ (s, i, si) =
        if si == 2
        then JUDGE (setnth (i, s, 1))
        else if si == 1
        then work -> pass ! i -> FRAME (setnth (i, s, 0))
        else work -> JUDGE (s)
```

When a pass signal has been indicated, we examine the new values of all the counters in s to see if they are now all zero. (This uses the FDR set operator.) If this is the case, further tasks are ignored after a comparison, and the end-of-frame synchronization may occur. Otherwise the sub-system returns to its initial state.

```
FRAME (s) =
        if set (s) == { 0 }
        then (sync -> COMMS) [] task ? any -> work -> FRAME (s)
        else JUDGE (s)
```