NASA Technical Memorandum 108784

Reactive System Verification Case Study—Fault-Tolerant Transputer Communication

D. Francis Crane and Philip J. Hamory

September 1993

National Aeronautics and Space Administration

NASA Technical Memorandum 108784

Reactive System Verification Case Study—Fault-Tolerant Transputer Communication

D. Francis Crane and Philip J. Hamory, Ames Research Center, Moffett Field, California

September 1993

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035-1000

Summary

A reactive program is one which engages in an ongoing interaction with its environment. A system which is controlled by an embedded reactive program is called a reactive system. Examples of reactive systems are aircraft flight management systems, bank automatic teller machine (ATM) networks, airline reservation systems and computer operating systems. Reactive systems are often naturally modeled (for logical design purposes) as a composition of autonomous processes which progress concurrently and which communicate to share information and/or to coordinate activities.

Formal (i.e., mathematical) frameworks for system verification are tools used to increase the users' confidence that a system design satisfies its specification. A framework for reactive system verification includes formal languages for system modeling and for behavior specification and decision procedures and/or proof-systems for verifying that the system model satisfies the system specifications.

In the study reported here, using the Ostroff framework for reactive system verification, an approach to achieving fault-tolerant communication between transputers was shown to be effective. The key components of the design, the decoupler processes, may be viewed as discrete-event-controllers introduced to constrain system behavior such that system specifications are satisfied.

The Ostroff framework was also effective. The expressiveness of the modeling language permitted construction of a faithful model of the transputer network. The relevant specifications were readily expressed in the specification language. The set of decision procedures provided was adequate to verify the specifications of interest.

The need for improved support for system behavior visualization is emphasized.

Introduction

Computer programs can be classified as either *transformational* or *reactive*. Transformational programs, the more common type, are typically designed to transform data via appropriate algorithms and to then output the results of the computation and terminate. First Order Logic (ref. 1) is routinely used to specify and to reason about the correctness of transformational programs. A reactive program is one that engages in an ongoing interaction with its environment (ref. 2). A system that is controlled by an embedded reactive program is called a reactive system. Examples of reactive systems are aircraft flight management systems, bank automatic teller machine (ATM) networks, airline reservation systems, and computer operating systems. Reactive systems are often naturally modeled (for logical design purposes) as a composition of autonomous processes which progress concurrently and which communicate to share information and/or to coordinate activities. Reactive systems are nondeterministic in that the sequence of events is not specified but depends on actions of the environment. Reactive system specifications often include response time requirements.

These reactive system process characteristics (autonomous, concurrent, communicating, nondeterministic, and time sensitive) have forced the development of new approaches to verify that a reactive system satisfies its specification. As noted by Alur (ref. 3), "The number of formalisms that purportedly facilitate the modeling, specifying and proving of timing properties for reactive systems has exploded over the past few years." The diversity of process communication and coordination constructs and the variety of specifications of interest have contributed to this profusion of frameworks. The features required to further improve next-generation frameworks can best be determined through use and evaluation of currently available frameworks in many diverse applications. One objective for this report is to contribute to that evolutionary process.

The framework chosen for the analysis of a particular system must allow faithful modeling of essential system features in order to reliably infer system behavior from model behavior. In the study reported here, a framework developed by Ostroff was applied to verify an approach to achieve fault-tolerant transputer communication. In the following sections, we outline the Ostroff framework, review the approach to fault-tolerant transputer communication verified, describe the Transputer Network Model, and discuss verification procedures and verification results. The need for improved support for system behavior visualization is emphasized.

The Ostroff Framework

Formal (i.e., mathematical) frameworks for system verification are tools used to increase the users' confidence that a system design satisfies its specification. A framework for reactive system verification includes formal languages for system modeling and for behavior specification and decision procedures and/or proof-systems for verifying that the system model satisfies the system specifications. Ostroff's book (ref. 4) should be consulted for a comprehensive description of the framework used in this study (hereinafter referred to as the Framework). The description here is informal and necessarily incomplete.

A system is modeled as a *composition* of autonomous, concurrent, communicating processes. Each process is represented by a diagram. The elements of the diagram are nodes and labeled, directed edges which connect nodes and which model process transitions. For each process an activity or control variable, A_V, is defined which ranges over the process nodes to indicate the location of *control* in the process.

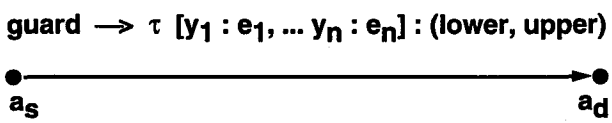
We next review two types of transition which will be needed to model the transputer network. An *assignment* transition is illustrated in figure 1.

The transition τ is *enabled* if *control* is at a_S ($A_V = a_S$) and if **guard** evaluates to **TRUE**. Enabled transitions are *held* for at least **lower** ticks of the external (conceptual) clock and must occur no later then **upper** ticks of the clock. If the enabled transition τ is *taken*, then A_V will be assigned the value a_d and the variables $y_1,..., y_n$ will be assigned the values of the expressions $e_1,...,e_n$, respectively . If a guard is missing, it is assumed to be **TRUE**. If the list of variables is missing, then no variables are assigned values by the transition. If the time bounds are missing, they are assumed to be (**lower**: 0, **upper**: infinity), i.e., the transition is neither held nor forced.

Processes communicate via named *channels* in order to either transfer information or coordinate activities. A *synchronous communication* transition is illustrated in figure 2.

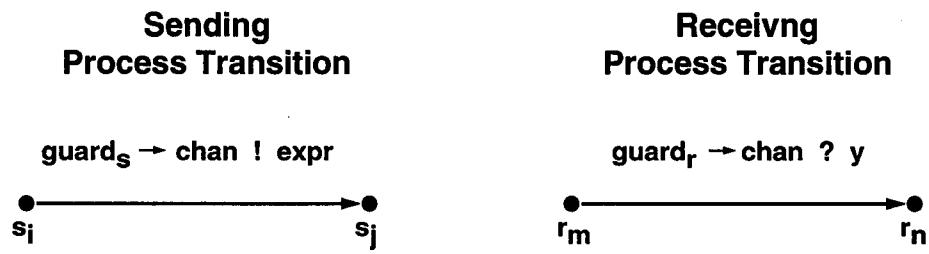
The meaning of the transition label "**chan ! expr**" is: if this transition is taken, then the value of the expression "expr" will be sent on channel "chan." The meaning of the transition label "**chan ? y**" is: if this transition is taken, then the value received on channel "chan" will be assigned to the variable "y." Communication is *synchronous*, i.e., enabled only if *matching* (same channel) transitions in both *sending* and *receiving* processes are *simultaneously* enabled. The first process to reach a ***send*** or ***receive*** transition will *block*, i.e., suspend activity, until the matching transition is also enabled. If an enabled communication transition is taken, the variable assignment described is made and then both processes continue independently.

A system *behavior* is a sequence of states wherein the initial state satisfies an initial condition specification and where following states are reached by taking an enabled transition in any component process. When transitions in a number of processes are enabled, the next transition taken is chosen nondeterministically. (The failure condition in which none of the component processes can progress because all transitions are disabled is called deadlock.) A system is said to satisfy a specification if all possible system behaviors satisfy the specification. The Framework specification language and decision procedures are described in a later section.



a_S	source node	$y_1, ... y_n$	variables
a_d	destination node	$e_1, ... e_n$	expressions
τ	transition label	**lower**	lower time bound
guard	boolean expression	**upper**	upper time bound

Figure 1. Assignment transition syntax.



guard$_S$, guard$_r$	boolean expressions	**!**	sending process identifier
s_i, r_m	source nodes	**?**	receiving process identifier
s_j, r_n	destination nodes	**expr**	expression
chan	communication channel	**y**	process variable

Figure 2. Synchronous communication transition syntax.

Fault-Tolerant Transputer Communication

A transputer is a very large scale integration (VLSI) device which combines on a single silicon chip—a processor, memory for program storage, hardware-timers, and communication controllers which permit direct synchronous communication with other transputers (ref. 5). Networks of transputers have been used to implement a wide variety of reactive systems including systems for (a) robot guidance and control (ref. 6), (b) piloted-helicopter simulation (ref. 7), and (c) signal processing (ref. 8). Approaches to achieve fault-tolerant communication between transputers were investigated in connection with a proposed aircraft application. Recall from the discussion of synchronous communication that a process which is *ready-to-send* will block until the matching process is *ready-to-receive*. If there is only a single physical channel between two transputers and that channel fails, then a process will block if it attempts to ***send*** on the failed channel. A system that depends on timely communication over the failed channel will fail.

One cannot achieve fault-tolerant communication between processes on different transputers by simply connecting a second physical channel *directly* between the processes and routinely sending all data over both channels. The sending process will block when it attempts to send on a *failed* channel even though the other channel is fully functional. An approach which does (as will be shown) provide fault-tolerant communication between a process PRODUCER executing on one transputer and a process CONSUMER executing on another is outlined in figure 3.

The key feature of the design is that two concurrent *decoupler* processes (DECOUPLER 1 and DECOUPLER 2) are defined on Transputer 1, each of which communicates with PRODUCER over two *internal* channels and with CONSUMER over a *physical* channel. (Internal channels, used to communicate between processes on the same transputer, are implemented in software.) DECOUPLER 1 continuously loops through a sequence of three synchronous communications:

1. Input data on internal channel **out1** from PRODUCER
2. Output data on physical channel **send1** to CONSUMER
3. Signal PRODUCER on internal channel **status1**

DECOUPLER 2 continuously loops through a similar sequence of three synchronous communications using channels **out2, send2,** and **status2.** When both physical channels are operational, PRODUCER sends all information to CONSUMER over both physical channels. If physical channel **send1** fails, then DECOUPLER 1 will block when it next attempts to use **send1.** However, PRODUCER will detect (infer) that DECOUPLER 1 is blocked if the signal on **status1** is not received within a prespecified time. Thereafter, PRODUCER will continue to communicate over the intact physical channel. The decoupler processes are effectively *discrete-event-controllers* introduced to constrain system behavior such that system specifications are satisfied.

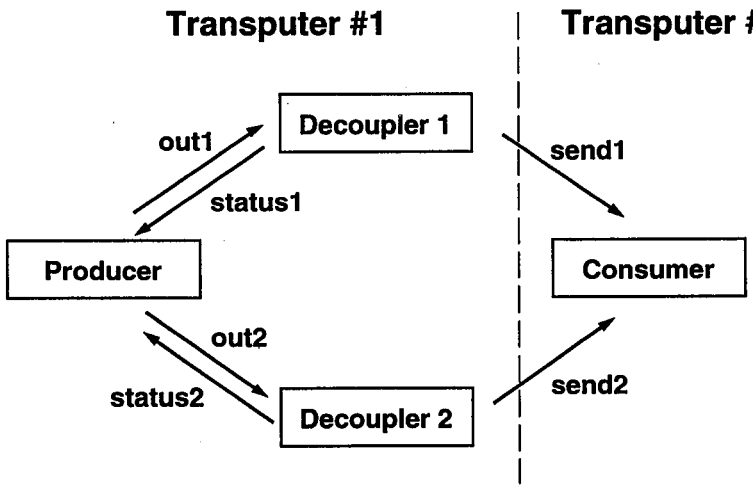


*Figure 3. Sketch illustrating the concurrent processes PRODUCER, DECOUPLER 1, DECOUPLER 2 and CONSUMER and the communication channels (**out1, out2, send1, send2, status1, status2**) connecting the processes. The decoupler processes are effectively discrete-event-controllers introduced to ensure that communication between PRODUCER and CONSUMER is not disrupted by failure of external channel **send1** or **send2**.*

The Transputer Network Model

OCCAM is the name of a concurrent programming language used to program transputers and transputer networks (ref. 9). To verify the approach to fault-tolerant transputer communication outlined above, an OCCAM implementation of the approach was first translated into the Framework diagram language representation shown in figure 4. A faithful translation was possible because both languages view systems as a composition of autonomous, concurrent, communicating processes and each OCCAM construct was expressible in the diagram language. *In particular, the semantics of the synchronous communication construct in each language was identical.*

The maximum size of the composite-system state space is an exponential function of the number of processes. Therefore, when attempting verification, it is important to simplify the system model by "abstracting away" unessential detail. Four such simplifications, which taken together reduce the size of the state space by many orders of magnitude, are incorporated into figure 4 and described next.

Focus on Process Communication Logic

The process communication logic is embedded in a simple, cyclic PRODUCER-CONSUMER system (fig. 3). The single transitions, **produce** in PRODUCER and **consume** in CONSUMER, represent the "other" activities of the communicating processes which typically include complex computations and communication with other transputers over other channels.

Simplify Data Structures

OCCAM channel protocol declarations permit communication of complex data structures. Data structure details are irrelevant when verifying OCCAM-level process communication logic because autonomous, lower-level controllers manage the physical data transfer. In figure 4, each communication transfers a single integer.

Project Behavior Using Logical Variables

An essential aspect of the design is the fact that, unlike a *sending* process which blocks until a matching receiving process is enabled, OCCAM semantics permit a *receiving* process to start a hardware-timer and to take a default action if the expected communication is not received before the timer "times out." When the external channels are functional, these *time-out* transitions are never taken. The logical variable *Fail1* (*Fail2*) is used in the *guard* of the **send1** (**send2**) channel time-out transitions to eliminate the time-out transitions from the *reachability graph* (described in the next section) when external channel **send1** (**send2**) is intact. Effectively we enhance *system behavior visualization* by obtaining a *projection* of relevant behavior.

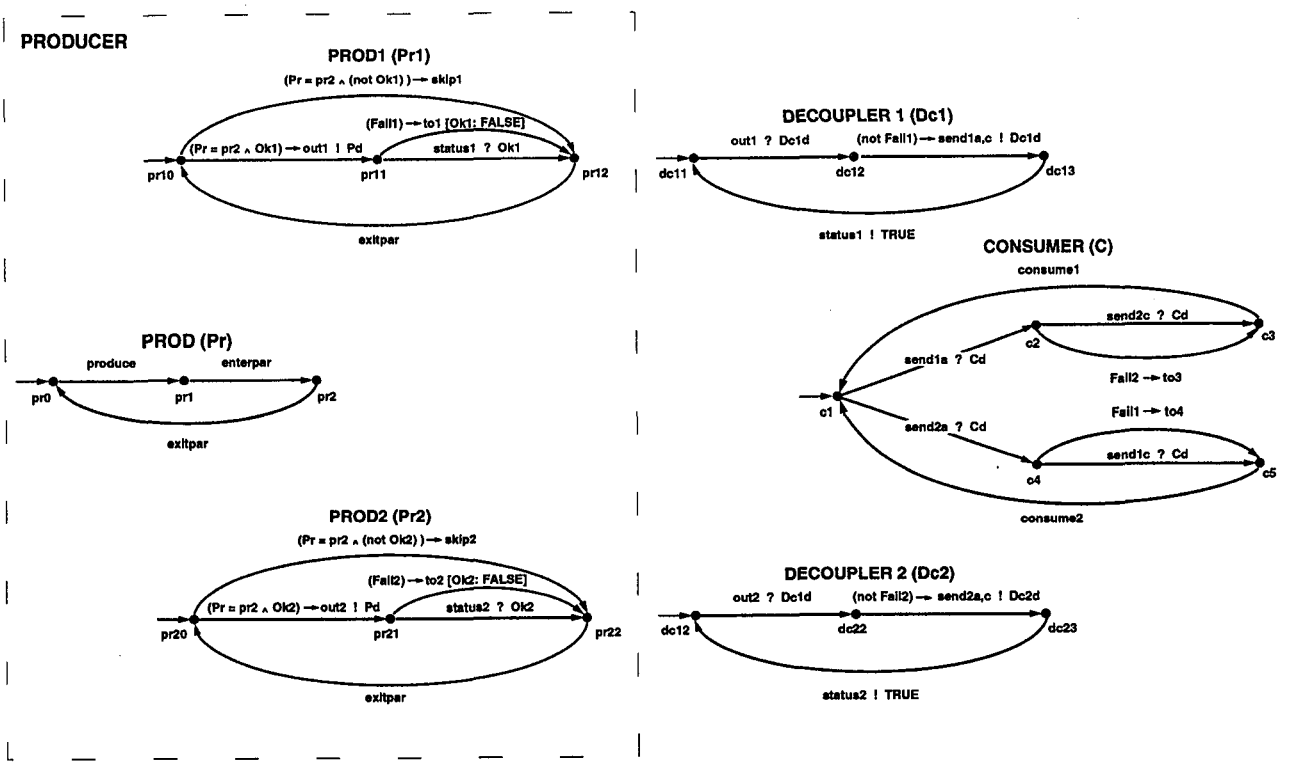


Figure 4. Ostroff diagram language representation of PRODUCER, DECOUPLER 1, DECOUPLER 2, and CONSUMER processes. The PRODUCER process is modeled as a composition of concurrent processes PROD, PROD1, and PROD2. The name of the Activity variable for each process is shown in parentheses following the process name. The initial value of the Activity variable for each process is indicated by an arrow (→). The initial value for all data variables (Pd, Dc1d, Dc2d, Cd) is zero. The transition labeled ***exitpar, which*** *occurs in PROD, PROD1, and PROD2, is an example of an interaction transition. The interaction* ***exitpar*** *is enabled when Pr = pr2, Pr1 = pr12, and Pr2 = pr22. If* ***exitpar*** *is taken, the processes PROD, PROD1, and PROD2 progress simultaneously. The transition label* ***send1a,c*** *means there are two transitions (****send1a, send1****) connecting the nodes.*

Simplify Hardware-Timer Details

Because here we verify only *qualitative* temporal logic specifications, the upper time bound on transitions that model hardware-timers are set to unity when verifying *response* properties.

Verification Procedures and Results

For finite-state systems, the Framework provides software which uses the component process models to compute a system *reachability graph* and *decision procedures* which use the graph structure in evaluating the validity of certain system specifications. A reachability graph is a list of vertices and a list of edges connecting vertices that summarize possible system behavior. Graph *vertices* represent system states, and graph *edges* represent transitions which change system state. A *behavior* of the system is a *path* (a sequence of states) in the reachability graph which starts at a state satisfying an initial condition specification. A system satisfies a specification if all possible behaviors satisfy the specification. We present results for two cases—the *Normal Operation* case and the *External Channel Failure* case.

Normal Operation Case Results

In normal operation of the transputer network modeled by figure 4, both external channels between transputers are functional. The reachability graph of the system for this case was *manually* diagrammed and is shown as figure 5.

The diagram is relatively simple because process model details irrelevant to verification of fault-tolerant communication have been abstracted away as described earlier. In this section, we rely heavily on this diagram in order to emphasize the usefulness of this *system-behavior-visualization* aid. For conciseness we refer to a diagram of a reachability graph as a Graph.

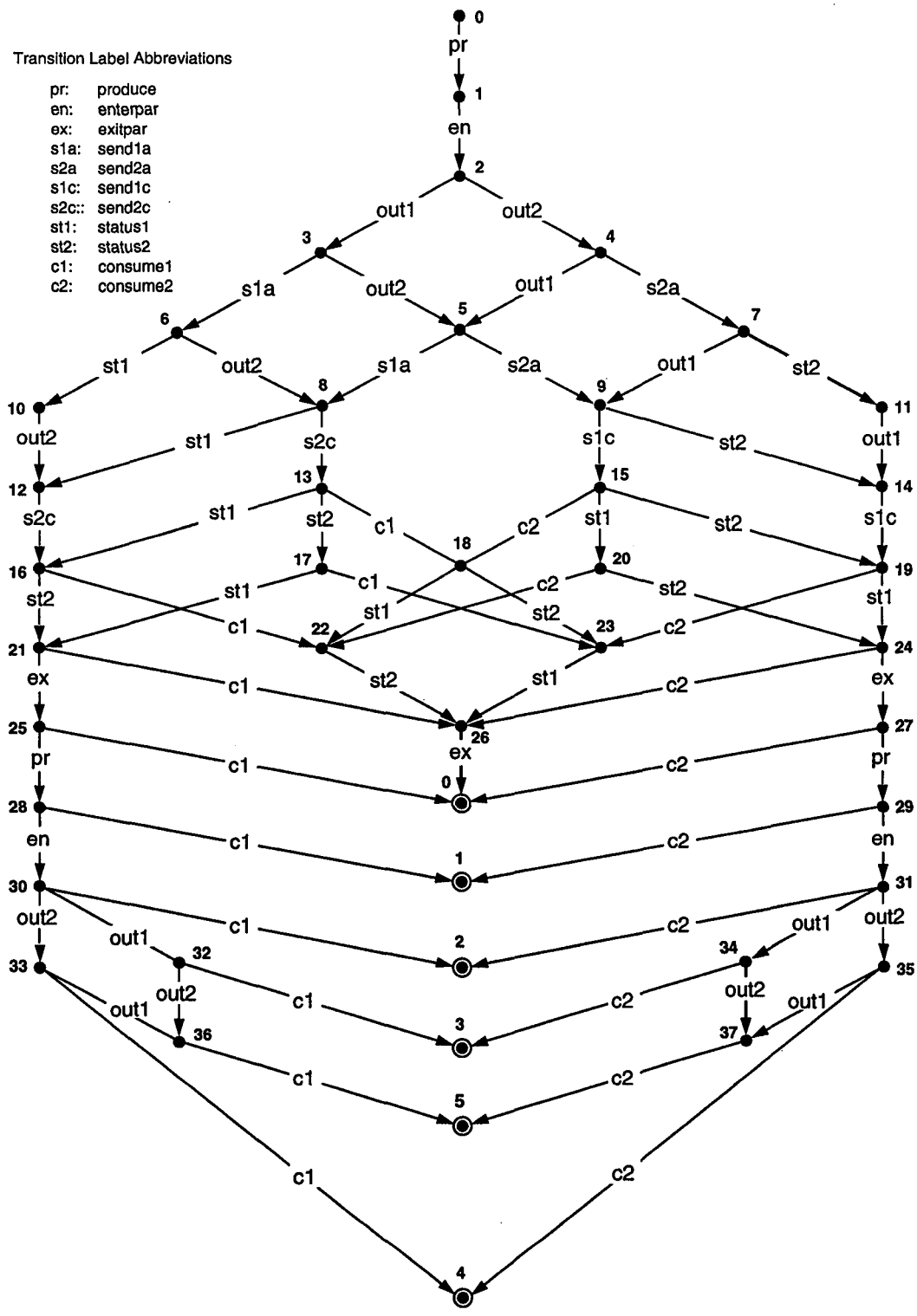


Figure 5. Reachability Graph, Normal Operation Case. As described in the text this Graph is a projection *of system behavior in that Timer transitions (never taken in Normal Operation) are suppressed in order to enhance system behavior visualization. In order to eliminate clutter resulting from long lines connecting vertices, some nodes are repeated. Repeated vertices are circled. The vertex number uniquely identifies the vertex.*

Important system characteristics are evident in figure 5:

The system is symmetric. The symmetry of the Graph reflects the symmetry in the component processes with regard to use of the communication channels between processes. (During a modeling effort, absence of expected symmetry or regularity is often a clear indication of a modeling error.)

The system is nondeterministic. Many states may be exited by several transitions—any one of which can be chosen in a particular cycle. Transitions from the component processes *interleave*, indicating the cooperation among the processes to transfer data. *Unanticipated interleaving often results in undesirable system behavior.*

When, as in this case, the reachability graph is relatively simple, certain system specifications can be verified by visual inspection of the Graph. The relevant specifications are determined by considering what can go wrong. The fact that communication is synchronous introduces the possibility of *deadlock* if process communication logic is flawed. The fact that all data are sent via two autonomous decoupler processes introduces the possibility that data may arrive at the CONSUMER process "out of order."

(In the following paragraphs, the symbols **S1, S2,** etc., are specification labels.)

Inspection of figure 5 will confirm that:

S1 *The system does not deadlock—*

because every state has exiting transitions.

S2 *All data produced are sent over both external channels in the order produced—*

because following each **produce** transition, both **send1** and **send2** transitions precede the next **produce** transition.

S3 *All data are consumed in the order sent—*

because following transmission of data over both channel **send1** and **send2**, a **consume** transition precedes the next occurrence of a **send1** or **send2** transition.

Together **S2** and **S3** imply that although the data are transmitted via two autonomous decoupler processes—

S4 All data produced are consumed in the order produced.

The insight provided by the Graph is also very important when attempting to write formal specifications in preparation for using the Framework decision procedures.

An "obvious" specification for temporal-ordering of the data is:

S5 Following a **produce** transition, a **consume** transition precedes the next **produce** transition.

Specification **S5** implies that data are consumed in the order produced. However, the Graph clearly shows that specification **S5** is unnecessarily restrictive (reference node 25). That specification would also be impossible to implement (without compromising the fault-tolerance objective) because the PRODUCER process has no information with regard to the status of the CONSUMER process. In the next section, the decision procedures are applied to verify similar properties.

External Channel Failure Case

We begin with a brief review of the Framework specification language and decision procedures. The Framework specification language is a Temporal Logic in which many important reactive system properties can be expressed. Temporal logic specifications are interpreted over system behaviors (i.e., sequences of reachable states) which are summarized by a system reachability graph. A system satisfies a temporal logic specification if all possible behaviors satisfy the specification. Discussion of temporal logic is beyond the scope of this report; instead, we include (necessarily) imprecise English language interpretations of the temporal logic expressions used. We next review the three classes (*safety*, *precedence*, and *response*) of Temporal Logic specifications that we will need.

A *safety* specification is conventionally expressed in the form

S6 $\psi_1 \rightarrow \Box\, \psi_2$

read: if ψ_1, then *henceforth* ψ_2 where ψ_1 and ψ_2 are state-formulas.

A system satisfies this specification if ψ_2 is **TRUE** for *all* states following any state for which ψ_1 is **TRUE**.

Specifications involving *temporal ordering* of transitions can be expressed using the temporal operator **P** (*precedes*) as in

S7 $\psi_1 \rightarrow \psi_2\, \mathbf{P}\, \psi_3$

read: if ψ_1, then ψ_2 *precedes* ψ_3 where ψ_1, ψ_2, and ψ_3 are state-formulas. A system satisfies this specification if following any state in which ψ_1 is **TRUE**—a state in which ψ_2 is **TRUE** precedes a state in which ψ_3 is **TRUE**.

A *response* specification is of the form

S8 $\psi_1 \rightarrow \Diamond \psi_2$

read: if ψ_1, then *eventually* ψ_2 where ψ_1 and ψ_2 are state-formulas.

A system satisfies this specification if following any state in which ψ_1 is **TRUE**—a state in which ψ_2 is **TRUE** is eventually reached.

The Framework provides *decision procedures* for safety, precedence, and response class specifications. The decision procedures use a system reachability graph, which summarizes possible system behavior, in evaluating specification validity. When a decision procedure for a class of specifications is invoked to verify a specification of the class, the decision procedure always terminates and either confirms the specification validity or provides information regarding the state(s) and transition(s) which violate the specification.

In the following paragraphs, we apply the specification language and decision procedures to verify that fault-tolerant communication between transputers is achieved. Specifically, we verify that *after failure of an external channel between transputers*:

S9 The system does not deadlock.

S10 All data are transferred between transputers in the correct temporal order.

The variables *Fail1* and *Fail2* provide a convenient way to introduce an external channel failure. Referring to figure 4, when *Fail1* is assigned the value **TRUE**, the DECOUPLER 1: **send1** transition is disabled which effectively models channel **send1** failure. The Graph (i.e., the diagram of the reachability graph) for this case is shown as figure 6.

The Graph includes both the "transient" system behavior in the cycle immediately following external-channel **send1** failure and the behavior in the cycles thereafter. We next express the informal specifications **S9** and **S10** in terms of safety, precedence, and response class specifications and then invoke the appropriate decision procedure to check specification validity.

As noted earlier, a system is said to be *deadlocked* if it is in a state in which no transition (other than the clock transition) is enabled. The system was verified to be deadlock-free by invoking the safety decision procedure to verify

S11 **initial** $\rightarrow \Box$ **((enabled** τ**) and (**$\tau \neq$ **Tick))**

i.e., following a state which satisfies the initial condition specification, some transition (other than the clock transition) is enabled in every reachable state.

Using the *precedence* decision procedure, we verified

S12
after_**produce** $\rightarrow$ **(Next = send2) P (Next = produce)**

i.e., after a transition which produces data, the data are sent before more data are produced (**Next** is the next-transition-taken variable)

and

S13
after_**send2** $\rightarrow$ **(Next = CONSUME) P (Next = send2)**

i.e., after a transition which sends data, those data are consumed before more data are sent.

Using the response decision procedure, we verified

S14 after_**produce** $\rightarrow \Diamond$ after_**consume**

i.e., all data produced are eventually consumed. The upper time bound for all transitions was set to unity in computing the more-complex reachability graph (not shown) used to verify S14.

Validity of specifications **S11, S12, S13, and S14** implies that fault-tolerant communication between transputers is achieved. After failure of an external channel between transputers—the system does not deadlock and all data are transferred between transputers in the correct temporal order.

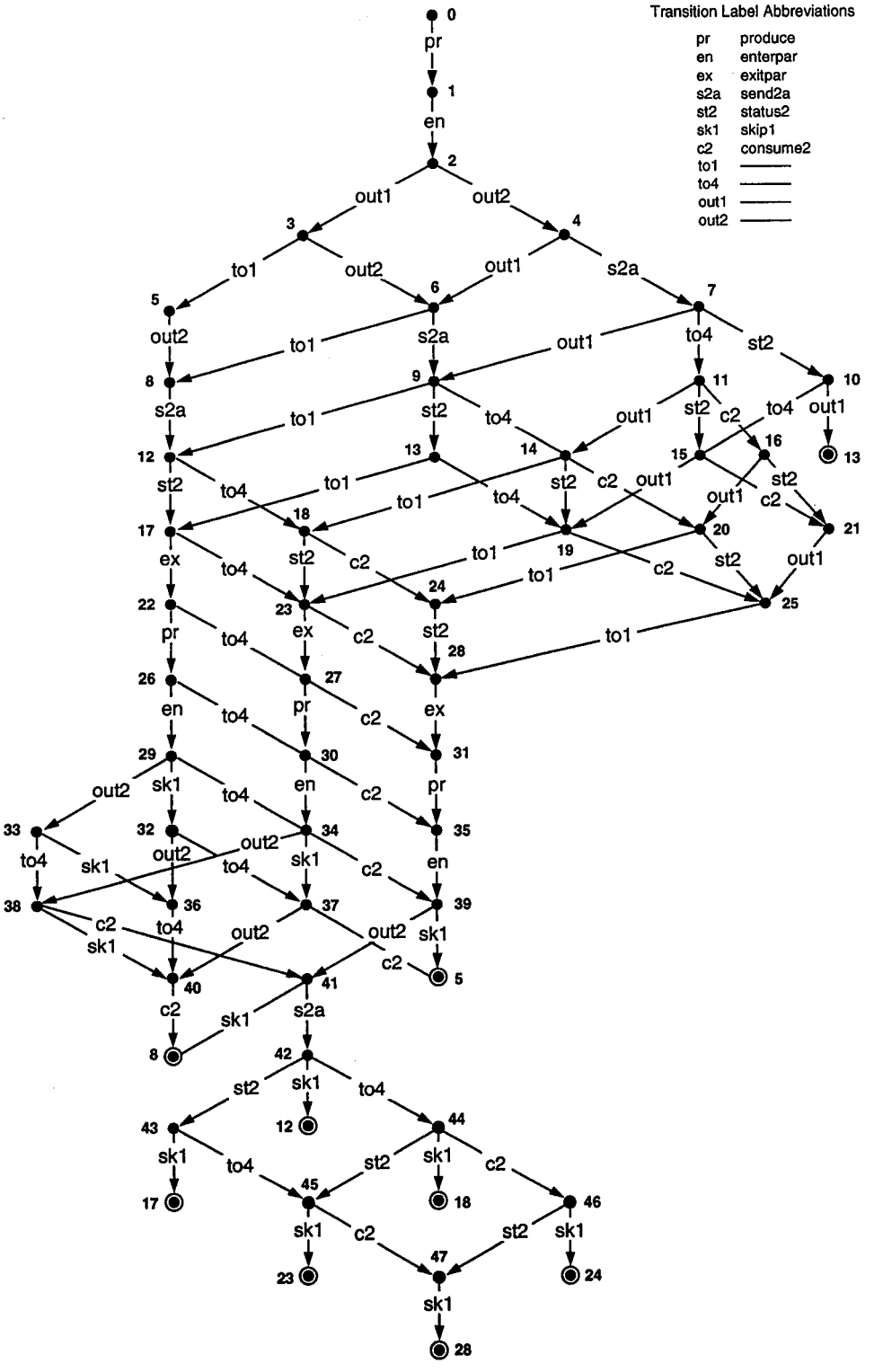


*Figure 6. Reachability Graph, External Channel (**send1**) Failure Case. This Graph is a* projection *of system behavior in that **send2** channel timer transitions are suppressed in order to enhance system behavior visualization. In order to eliminate clutter resulting from long lines connecting vertices, some vertices are repeated. Repeated vertices are circled. The vertex number uniquely identifies the vertex.*

Concluding Remarks

In the preceding, using the Ostroff framework for reactive system verification, an approach to achieving fault-tolerant communication between transputers was shown to be effective. The key components of the design, the decoupler processes, may be viewed as discrete-event-controllers introduced to constrain system behavior such that system specifications are satisfied.

The Ostroff framework was also effective. The expressiveness of the modeling language permitted construction of a faithful model of the transputer network. The relevant specifications were readily expressed in the specification language. The set of decision procedures provided was adequate to verify the specifications of interest (although decision procedures to verify more general classes of temporal logic specifications will often be useful or necessary).

However, the Ostroff framework and other current generation frameworks for reactive system verification are particularly weak in one very important dimension, namely, support for system behavior visualization. (The importance of system behavior visualization during the verification process was emphasized in the section discussing Normal Operation Case results.) "Inability to visualize system behavior" is a factor restricting current applications to small, safety-critical portions of complex systems. As a first step, software tools enabling one to interactively construct, to browse, and to compare reachability-graph diagrams are needed. Manual construction of these basic visualization aids is an extremely tedious task. There is great opportunity for innovation with regard to system behavior visualization tools. For example, in a related context, an approach wherein sequences of transitions are mapped into higher-level transitions improved behavior visualization (ref. 10). The surveys by Ostroff (ref. 11), Alur and Henzinger (ref. 3), and Scholfield (ref. 12) describe the vigorous, current research effort that is directed at developing more powerful frameworks for reactive system verification.

References

1. Manna, Z.; and Waldinger, R.: Logical Basis for Computer Programming. Addison-Wesley,1989.
2. Manna, Z.; and Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, 1991.
3. Alur, R.; and Henzinger, T. A.: Logics and Models of Real Time: A Survey. In REX Workshop—Real Time: Theory in Practice, LNCS. Springer-Verlag, 1992.
4. Ostroff, J. S.: Temporal Logic for Real-Time Systems. Advanced Software Development Series. Research Studies Press Limited (distributed by John Wiley and Sons), England, 1989.
5. INMOS staff. Communicating Process Architecture. Prentice Hall, 1988.
6. Barnes, D. P.; Downes, C. G.; and Gray, J. O.: The Real Time Control of a Hexapodal Robot Using Multiple Transputers. In Proceedings of the Third International Conference on Applications of Transputers. IOS Press, 1991.
7. Lawes, S. T.; Clarke, T.; and Taylor, P.: Transputer Based Real-Time Simulation of Helicopter Dynamics for Advanced Flight Control Applications. In Proceedings of the Third International Conference on Applications of Transputers. IOS Press, 1991.
8. Mandal, M.; Garg, H. K.; Matieda, I. C.; Mishra, A.; Basu, S. K.; Majumder, K. L.; Udpikar, V.; and Kaushal, A,: Implementation of Synthetic Aperture Radar Processor on a Transputer Based Parallel Machine. In Proceedings of the Third International Conference on Applications of Transputers. IOS Press, 1991.
9. Galletly, J.: OCCAM 2. Pitman Publishing, 1990.
10. Gennart, B. A.; and Luckham, D. C.: Validating Discrete Event Simulations Using Event Pattern Mappings. In Proceedings 29th ACM/IEEE Design Automation Conference. IEEE Computer Society Press, 1992.
11. Ostroff, J. S.: Formal Methods for the Specification and Design of Real-Time Safety Critical Systems. Journal of Systems and Software, April 1992.
12. Scholfield, D. J.: The Formal Development of Real-time Systems. Technical Report, Dept. of Computer Science, University of York, England, 1990.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1993	Technical Memorandum

4. TITLE AND SUBTITLE

Reactive System Verification Case Study—Fault-Tolerant Transputer Communication

5. FUNDING NUMBERS

505-64-52

6. AUTHOR(S)

D. Francis Crane and Philip J. Hamory

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Ames Research Center
Moffett Field, CA 94035-1000

8. PERFORMING ORGANIZATION REPORT NUMBER

A-93103

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA TM-108784

11. SUPPLEMENTARY NOTES

Point of Contact: D. Francis Crane, Ames Research Center, MS 210-3, Moffett Field, CA 94035-1000
(415) 604-1434

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified — Unlimited
Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

A reactive program is one which engages in an ongoing interaction with its environment. A system which is controlled by an embedded reactive program is called a reactive system. Examples of reactive systems are aircraft flight management systems, bank automatic teller machine (ATM) networks, airline reservation systems and computer operating systems. Reactive systems are often naturally modeled (for logical design purposes) as a composition of autonomous processes which progress concurrently and which communicate to share information and/or to coordinate activities.

Formal (i.e., mathematical) frameworks for system verification are tools used to increase the users' confidence that a system design satisfies its specification. A framework for reactive system verification includes formal languages for system modeling and for behavior specification and decision procedures and/or proof-systems for verifying that the system model satisfies the system specifications.

In the study reported here, using the Ostroff framework for reactive system verification, an approach to achieving fault-tolerant communication between transputers was shown to be effective. The key components of the design, the decoupler processes, may be viewed as discrete-event-controllers introduced to constrain system behavior such that system specifications are satisfied.

The Ostroff framework was also effective. The expressiveness of the modeling language permitted construction of a faithful model of the transputer network. The relevant specifications were readily expressed in the specification language. The set of decision procedures provided was adequate to verify the specifications of interest.

The need for improved support for system behavior visualization is emphasized.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Reactive system verification, Discrete event systems, Parallel programming, Transputer	12
	16. PRICE CODE A02

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		