Argonne National Laboratory, with facilities in the states of Illinois and Idaho, is owned by the United States government, and operated by The University of Chicago under the provisions of a contract with the Department of Energy.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

This report has been reproduced from the best available copy.

Available from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161

Price: Printed Copy A03
Microfiche A01

Distribution Category:
Mathematics and Computer
Science (UC-405)

ANL-90/43

ANL--90/43

DE91 007144

ARGONNE NATIONAL LABORATORY
9700 S. Cass Avenue
Argonne, IL 60439–4801

Formal System Specifications – A Case Study of Three Diverse Representations

by

G. H. Chisholm, B. T. Smith,[1] and A. S. Wojcik[2]

The Mathematics and Computer Science
Division and The Reactor Analysis Division

December 1990

This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research and by the Reactor Core Technology subprogram of the Nuclear Energy Program, U.S. Department of Energy, under Contract W-31-109-ENG-38, and in part by the Strategic Defense Initiative Organization, Office of the Secretary of Defense, under PMA-2304

[1] Also of The University of New Mexico, Department of Computer Science, Albuquerque, NM 87131

[2] Also of Michigan State University, Department of Computer Science, East Lansing, MI 48864

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Contents

List of Figures iv

Abstract 1

I Introduction 2

II Background 3

II.A Description of a reactor control subsystem for flow protection 4

II.B Overview of a fault-tolerant processor 5

II.C Hardware verification of fault tolerance 8

II.D Software specification 9

II.D.1 Background 10

II.D.2 Starting point 10

II.D.2.a Function specification 10

II.D.2.b Abstract interpretation 10

II.D.3 Signal validation 11

II.D.3.a Function specification 11

II.D.3.b Abstract interpretation 11

II.D.4 Sensor validation 11

II.D.4.a Function specification 11

II.D.4.b Abstract interpretation 12

II.D.5 Generate trip 12

II.D.5.a Function specification 12

II.D.5.b Abstract interpretation 12

II.D.6 Vote on trip 12

II.D.6.a Function specification 12

II.D.6.b Abstract interpretation 12

II.D.7 Conclusion 13

III FAM Specification 13

III.A Introduction to FAM 13

III.B Details of specification 14

IV Z Specification 14
IV.A Introduction to Z 14
IV.B Objects and Types 16
IV.B.1 Sets and set types 16
IV.B.2 Tuples and Cartesian product types 16
IV.B.3 Bindings and schema types 16
IV.B.4 Relations and functions 17
IV.B.5 Properties and schemas 17
IV.B.6 Decorations 17
IV.B.7 Combining schemas 18
IV.B.8 Conclusion 18
IV.C Details of specification 18
V STATEMATE® Specification 21
V.A Introduction to STATEMATE 21
V.A.1 Background 21
V.A.2 Higraphs 21
V.A.3 Hyperedges 22
V.A.4 Discussion 22
V.B Details of specification 22
VI Summary and Conclusion 23
Bibliography 29

List of Figures

Figure 1 Block Diagram of Reactor Control Subsystem 4
Figure 2 Data Flow in the Four-Processor CSDL FTP 6
Figure 3 Simplified Representation of the Data Flow in the Four Processor CSDL FTP 8
Figure 4 Abstract Specification for the Application Program — FAM . . . 15
Figure 5 Schema – square 18
Figure 6 Schema – square′ 18
Figure 7 Z Specification for Determine Sensor Agreement 19
Figure 8 Z Specification for Generic Algorithm 20
Figure 9 Simple blobs and unique contours describe all identifiable sets . 21
Figure 10 STATEMATE specification — Level 1 — DL's associated with sensors 22
Figure 11 STATEMATE specification — Level 2 23
Figure 12 STATEMATE specification — Level 3 24
Figure 13 STATEMATE specification — Level 4 25
Figure 14 STATEMATE specification — Level 5 26
Figure 15 STATEMATE specification — Complete 27

Formal System Specifications – A Case Study of Three Diverse Representations

by

G. H. Chisholm, B. T. Smith, and A. S. Wojcik

Abstract

The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand. [Dijkstra 1972]

Increasingly, computers are being used to control life-critical systems. This trend is based on perceived cost advantages gained from the flexibility assured by programmability. However, this flexibility is often impacted by an increase in complexity that often results in unexpected performance — performance that has life-critical implications.

How does one raise the confidence level of a computer to that acceptable for life-critical applications? Extending Dijkstra's assertion to encompass the entirety of the implementation, i.e., including hardware and operating system software, we conceive an approach that results in designs that are correct with respect to critical performance requirements.

This approach may be described by a mathematician as a constructive proof of correctness in contrast to traditional methods for proving correctness, e.g., analysis (probabilistic risk assessment or failure modes and effects), simulation, or testing.

The concept of constructive proof of correctness or formal system specification is found in an area of research referred to as Formal Methods [1]. Formal Methods describes a broad area that spans formalization of VLSI designs, communication protocols, software functionality, microprocessors, and security systems.

A review of the literature was conducted to investigate the application of Formal Methods to formal system specification. The result of this review was the identification of three categories of methods that potentially applied. A case study was devised to compare representative methods from each category to ascertain their respective merits. The case study was based on an actual system intended for use in the control of an operating nuclear reactor. However, the specification defines an algorithm generalized for diverse control systems. This paper describes the case study.

I Introduction

Actually, the designer of a computer-based, life-critical system has two worries:

- *The system* **is not** *operating as specified, and*
- *the system* **is** *operating as specified.* [Chisholm 1988]

The initial aspect of this study concentrated on identifying Formal Methods (FM) that may apply to system specification. A review of the literature was conducted, and a number of articles were selected for a detailed review. [2, 3, 4, 5, 6, 7, 8, 9, 10, 11].

One result of the detailed review was the understanding that there are as many FM's as there are *Formal Methodists*. One way of classifying FM's is by their approach or underlying philosophy, i.e.,

1. analytical or correctness proving approach, e.g., Boyer-Moore logic, traces, formal software documentation, and Floyd-Hoare induction assertion method;
2. synthetic or specification approach, e.g., Z, VDM, FAM, and STATEMATE; and
3. process approach, e.g., Cleanroom.

The analytical or correctness-proving approach may be typified by proving that an implementation is commutable to its specification. This requires a complete and unambiguous specification, details respective to the implementation, and a formalism to establish commutation between specification and implementation. Commutation in this context implies a mathematical system of functions that maps one entity to another. The domain of the system of functions is the specification and the range is the implementation.

The decision to label the second approach as *synthetical* was based on the need to describe the orthogonality of the approaches. The synthetical approach concentrates on the specification phase of system design. That is, the objective is to provide a formalism for the development of complete and unambiguous specifications. These specifications may be defined at a level of abstraction that is removed from the concrete or implementation level.

The analytical approach requires that there be an implementation prior to initiation of the formal application. The synthetical approach facilitates multiple levels of abstraction. However, the decomposition between levels of the specification and its completeness complicates this approach.

Examples from the analytical approach demonstrate commutativity between a level of abstraction that is a level of refinement above the implementation. The primary difference between 1 and 2 is philosophical. Examples from the synthetical approach are typically high-level abstractions of the system, from either a property or a model standpoint.

Extensions of these approaches provide departures from this generalization. However, a review of the literature provided no evidence of the application of either category 1 or 2 to the problem of total system. That is, examples exist of application to hardware [12, 10] or software [4, 5, 2] but not to their composition.

The traditional methods that fit in the analytical category are applied to determine whether the design implementation conforms with a specification. The burden associated with application of these methods is twofold: the effort associated with the verification, and that associated with correcting errors discovered during verification. An insidious aspect is that the corrected system must then be verified, resulting in subsequent verification and correction, ad nauseum.

The process approach, e.g., IBM's Cleanroom, encompasses multiple disciplines and addresses the totality of the design process. This approach is directed toward large organizations that may commit the resources required to support diverse expertise, e.g., statistics, verification, and functional specification. This approach involves the synthesis of approaches to various aspects of the design process. For example, statistical analysis is applied to verification of software design. Two difficulties arise: first, the organizational commitment is large in that expertise in multiple disciplines is required and, second, the interfaces between the analysis resulting from the application of diverse methods provides additional complexity.

Two questions thus arise. First, is there adequate synergism when multiple methods are applied to the analysis of designs? Second, how does one assure that the system implements the specification properly? This, of course, applies to a system where correct operation is always required, say, for control of a nuclear reactor or some comparable life-critical system.

As stated above, the intent of the preliminary literature review to identify FM's that applied to the specification of systems. The analytical and process approaches were judged inadequate and overly complex respectively. A more detailed review of the literature concentrated on the synthetical approach to FM's.

A result of this detailed review was the understanding that the essence of an FM is the representation: How does one generate a specification within the framework of the formalism? The following is a list of classifications determined by representation:

1. Formal Languate, e.g., Z and VDM;
2. Visual, e.g., STATEMATE and 001; and
3. First Order Logic, e.g., FAM and B-M Logic.

The remainder of this paper describes the comparison of representatives from each category. This comparison is based upon using a single, well-understood (by the author) specification, and representing this specification by three methods. The intent of this comparison is to provide an understanding of the nature of each method. The organization of the description of this comparison consists of background information related to the system being specified followed by an informal description of the specification for the subsystem. This description introduces an abstraction for specifying the essential property of the system. The subsequent sections present three specifications. Each specification is preceded by a brief introduction to each method. The paper is concluded by a discussion of results of this case study into FM's.

II Background

Formal Methods:

- *Methods that add mathematical rigor to the development, analysis, and operation of computer systems and to applications based thereon,* [Neumann]

- *A form of applied mathematics, wherein one models the behavior of a physical device that is controlled by a computer program,* [Craigen]
- *A combination of deductive reasoning (mathematical thought) and inductive reasoning (deductive thought) applied to a determination of the validity of a formal description.* [Andrews]

II.A Description of a reactor control subsystem for flow protection

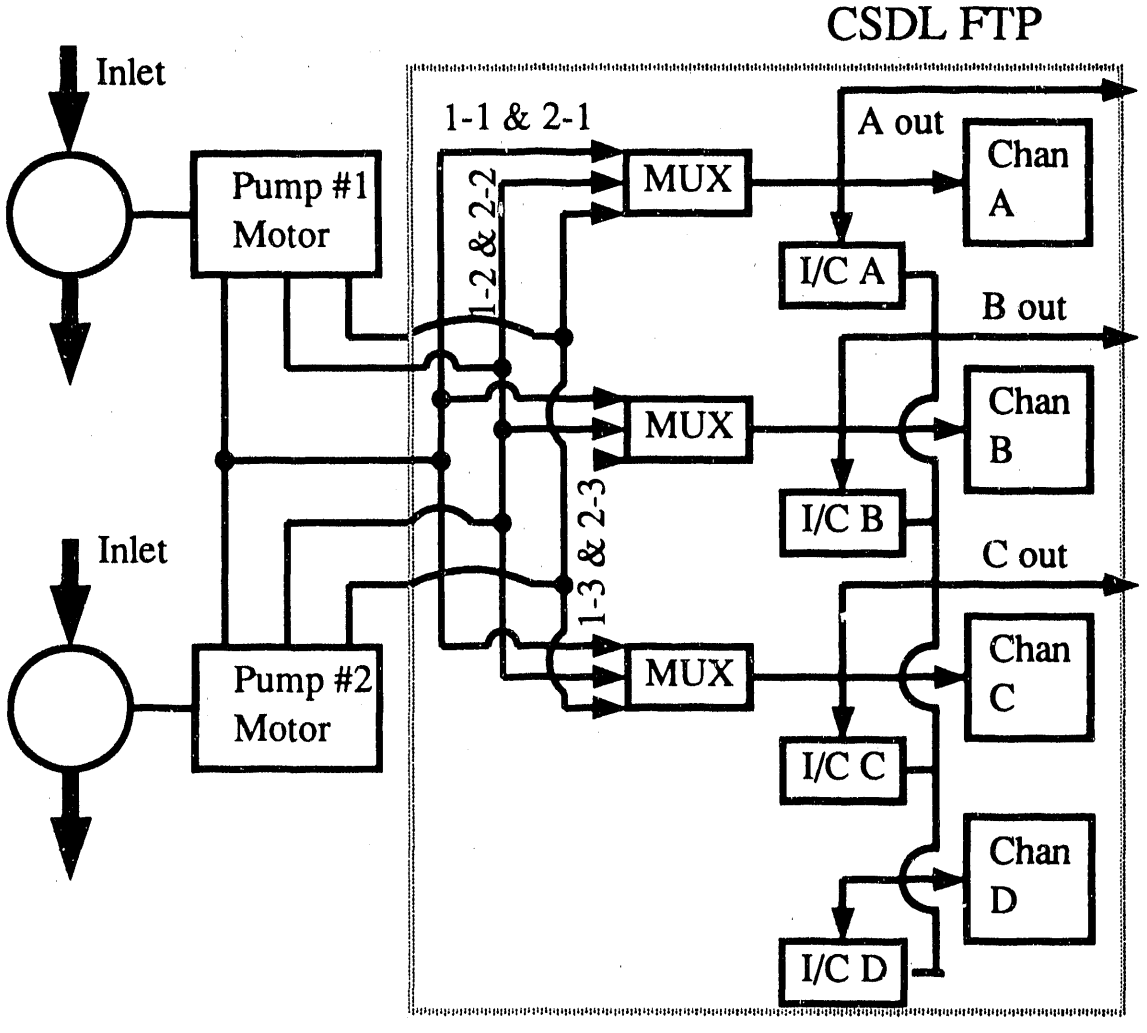

Figure 1 Block Diagram of Reactor Control Subsystem

Figure 1 is a block diagram of the reactor control subsystem that is the basis for this case study. The function of this subsystem is to monitor reactor coolant flow and, should the flow rate decrease below a predetermined level, provide a rapid reduction in reactor power. This rapid reduction in power is called a "TRIP"; that is, the nuclear reaction rate in the reactor core is reduced to a low level.

This reduction protects the reactor from overheating. Monitoring coolant flow is one of multiple system signals that provide similar protection for safe reactor operation. However,

operation of this subsystem is essential to safe reactor operation. As such, the system is required to be available 100% of the time that the reactor is operating. In addition, the system is required to perform its function with ultra-high reliability; the probability of failure is stated as 10^{-6} failures over a 1000–hour mission.

The pumps that propel the coolant around the reactor system are depicted in Figure 1 in the left half of the diagram. Two pumps are depicted in the diagram. Each pump motor (Pump #1 Motor, Pump #2 Motor) has been instrumented with three sensors. These sensors detect the revolution rate for the motors. Information about pump performance is used to correlate the number of revolutions to the coolant flow rates.

Three signals from each pump (e.g., 1–1, 1–2, 1–3) are connected to a multiplexor (MUX). Three separate multiplexors are used so that failure of one multiplexor affects a single signal only.

The multiplexor is an integral part of a fault-tolerant processor (FTP). For this subsystem, the FTP was designed by the Charles Stark Draper Laboratory (CSDL). The FTP consists of four identical channels that operate in very tight synchronization. Each channel of the FTP consists of a microprocessor (labeled Chan A, B, C, D), a multiplexor, an interstage/communicator (I/C), and an output (A out, B out, and C out).

This subsystem monitors reactor coolant flow. Upon detecting that the flow is below a predetermined level, the system generates a TRIP signal. An abstract description of this function is as follows:

1. Acquire data from six sensors,
2. Distribute date among the four channels,
3. Execute the application program, and
4. Output the required control.

The specification for the application program is the basis for this case study and will be described and formally stated in the latter sections of this paper.

II.B Overview of a fault-tolerant processor

The essence of fault-tolerant processors[4] is failure detection, isolation, and reconfiguration (FDIR). All failures must be detected to be tolerated. Toleration of detected failures requires that failures be isolated by the system and that the system reconfigure to a non-faulted configuration. Real time (i.e., cycles less than 100 milliseconds) restricts the amount of processing available for FDIR. The FTP design implements the mechanism for detecting and isolating failures in hardware. A brief description of this design follows.

Figure 2 illustrates the data flow for the FTP of Figure 1. The correlation between the FTP depicted in Figures 1 and 2, for Chan A, is

1. Chan A = Proc_a, Versa_a, Mem_a, I/O_a and VME_a
2. I/C A = Inter_a and Com_a.

[4] This section has been excerpted from [13].

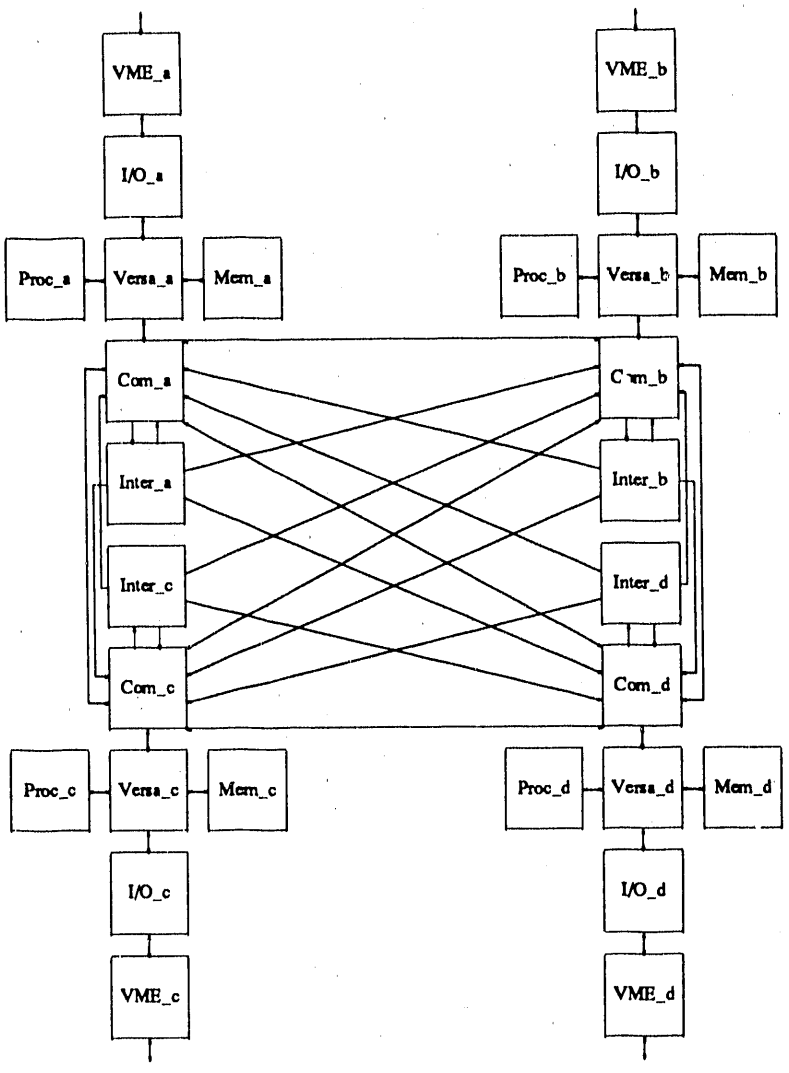

Figure 2 Data Flow in the Four-Processor CSDL FTP

The FTP [14] consists of four identical *nodes* connected through their communicators (Comm_) and interstage registers (Inter_) in an identical manner. The nodes are composed of off-the-shelf components and are identical, except for switches which identify each processor in a unique way. The software in each node is identical, including both the application software and operating system, and executes in synchrony in each node. Provided that every component in each node is fault-free and the clocks in each node remain synchronized, the nodes can be considered to be identical at all times, except for the setting of the identifier switches. Thus, from the viewpoint of the operation and programming of the application, the machine appears to be, and is, programmed as if it were a simplex system.

Although the FTP is designed to remain in synchrony, it is subject to failures which potentially invalidate this design claim. To tolerate such failures, the operating system schedules

tests at regular intervals to check that the data in the memory of each node is the same and that the behavior of the communication mechanisms between the nodes are operating correctly. Since the communicators vote on the data, bit by bit, a fault in a node can be detected. The voting is performed on a best two-out-of-three basis, with one of the four nodes masked from the voting process. The fourth (spare) processor is participating in all of the operations in synchrony with the three active nodes. Hence, at any given time it is known whether or not the spare is operational and that it has the same data as the other nodes. Provided that the spare is fault-free, it is ready to assume the role of any faulted node. The operating system in conjunction with the hardware will reconfigure the FTP so that the faulted node is masked out and replaced by the spare processor. Finally, synchrony of the node clocks is maintained at the hardware level by a special fault-tolerant clock.

Figure 2 shows the interconnection of four nodes, referred to as Node_A, Node_B, Node_C, and Node_D, with each node comprising a cluster of units, including a processor unit (Proc_) which is a Motorola 68000. The function of each component is discussed in [15, 16, 17, 14].

Figure 3 simplifies Figure 2 in the sense that the communicator in each node is merged with its processor, and only the data flow among the four nodes is represented. (The boxes represent the partition of the FTP into eight *fault-containment regions* (FCR's).

The focus of the design of the FTP is its ability to provide *congruent* data to a majority of the processors. Congruent means that the majority of the processors are guaranteed to be operating on the same data value if that data value has been distributed through the data network connecting the four nodes of the system. Congruent does not mean correct. In Figure 3, if Proc A distributes a data value to the other three nodes, the hardware voting elements that are built into the communicator units guarantee that the same voted data value will be stored by each processor. It may be that due to errors, the transmitted data value is corrupted so that the process of voting results in a wrong data value being accepted by each of the processors; but in this case, the data values accepted by the majority of the processors are congruent. Because a bit-by-bit vote occurs, they are the same values, although they may be incorrect. If a majority of the voted data is identical to the data that was initially sent by Proc A, the voted data is said to be *consistent.*

The claim of fault-tolerance for the FTP is linked to the manner in which the FTP operates. Each of the four processors is assumed to be executing the identical program. Further, all processors are executing the same instructions in lock step synchrony with one another. Data is transmitted bit serialy in the FTP. When a data value is computed by each of the processors, each distributes its value to the others via the data distribution network, referred to as a *voted exchange.* In this situation, the voting mechanism not only guarantees congruent data being stored by each processor, it results in identical data being stored even in the presence of a single failure in the system.

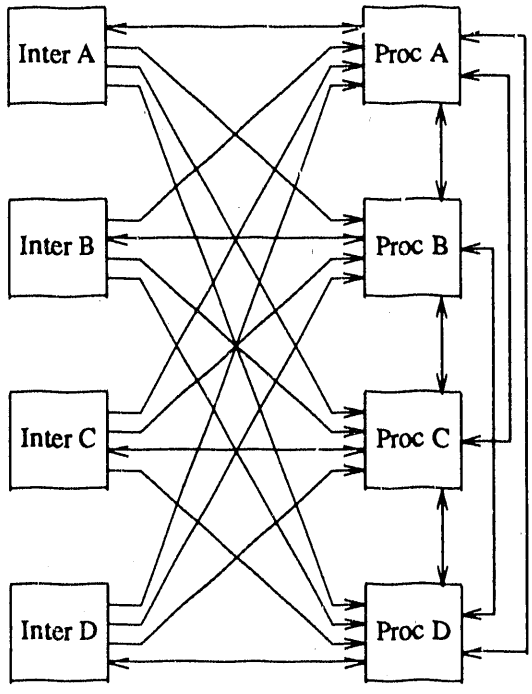

Figure 3 Simplified Representation of the Data Flow in the Four Processor CSDL FTP

The fact that single failures are tolerated by the system is dependent on the physical design of the FTP. Fault tolerance in the system means that, if the hardware in a single FCR malfunctions, at most a single error is generated out of the FCR. The single error could be the result of several failures within the faulted region. However, the failures still only result in the propagation of a single error to the other regions. This error is voted out, or masked, by the correctly functioning units in the system. The error is detected during the voting process, and the malfunctioning region is masked out of further voting until repair is accomplished.

II.C Hardware verification of fault tolerance

This section summarizes work reported in [15, 16, 17]. Specifically, this section identifies an abstraction used for analysis of the fault-tolerant properties of the FTP and presents the conclusions of the analysis.

The decision to use the data flow between FCR's for fault detection resulted in a very regular design. That is, there are two types of FCR's, Inter and Proc as shown in Figure 3. Fault

detection is based on a bit-by-bit comparison of data on initiation and receipt of a transmission. This comparison provides a dual function: 1) detection of faults and 2) assurance that each processor has exactly the same data as a result of the transmission. As fault detection is the result of data comparison, and data is generated from computation at each node, replicated data must reside at each node.

As mentioned above, fault tolerance is dependent on the detection of failures and on a sequential occurrence of failures. This design dependence led to an abstraction that allowed analysis of the FTP.

This abstraction, referred to as a dependency list (DL) [17] analysis, reduced the analytical burden significantly over that for traditional case analysis. A DL consists of the set of elements that make up an FCR. That is, the interstage FCR is a state machine that handles various aspects of the data flow. Its DL is a set that consists of each element shown on the schematic for this component. A determination of fault tolerance becomes a set theoretic solution: the intersection of a pairwise comparison of DL's results in empty lists. The interpretation of these empty sets is that there are no common elements between channels of the FTP. This implies that no single failure in one FCR will disrupt service in another FCR. Independence between channels is preserved, and fault tolerance holds.

II.D Software specification

Earlier the application program was briefly outlined. This section provides a more detailed description. This description will provide the basis for comparing three FM's from a representation standpoint.

The following sections describe individual modules that make up the application program. Each module performs a specific function and is described with an informal description of that function accompanied by an *abstract interpretation* [18] of that function.

That is; "An approximation to a value, such as the sign of an integer, is called an *abstraction*, and a computation over such abstract values is called an *abstract interpretation*".[5]

The abstract interpretation part of the specification is based on the DL abstraction and captures the essence of independence. It is a specification for a programming paradigm that preserves the fault tolerance of the underlying FTP and thus of the overall function of providing a flow trip signal for the reactor. For the purposes of this paper, the level of abstraction for the DL specification will be at a high level, and functionality refers to data manipulation in contrast to function.

A module may represent a large computational element. As in the hardware abstraction, an element is associated with an FCR. Only the computational FCR's are applicable to the software specification. Communication between elements is directed by data flow and coordinated by control flow. The essential property preserved by the specification is the independence established by the hardware proof.

From the viewpoint of FCR's, a module that performs computation on multiple parameters is interpreted in this abstraction as forming the union of the components associated with each

[5] Extracted from [18]

parameter. That is, a correct result depends on the fact that the components associated with each parameter being free of faults. This abstract interpretation ignores the functionality of the module. That is, irrespective of the correctness of the function, the data dependence has hardware dependence. Once this is established as being free of hardware induced failures, the issue of functionality may be addressed. This issue is the subject of another paper.

That is, a computational module that is initiated with data from two FCR's performs the equivalent of taking the union of the DL's associated with each data. In contrast, a module that compares two data, and makes a decision based on this comparison, performs the equivalent of forming the intersection of the associated DL's.

The next sections present a software specification that identifies modules by functionality and abstract interpretation.

II.D.1 Background

As noted in Section II, each processor executes identical software in synchrony with the other processors. The following description refers to execution on processor A exclusively, but implies identical operation on the others.

The FTP operating system is a single rate group (in contrast to other real-time systems having the ability to service jobs at multiple rates), multitasking environment. Foreground tasks are assumed to run to completion, and each executes within a predetermined time. These assumptions are essential to preserve tight synchrony between channels. A foreground manager provides context switching among foreground tasks. The application program runs as a foreground task.

II.D.2 Starting point

Upon a context switch to the application program, the operating system has provided an environment for continued execution. This environment assures that congruent data from each sensor resides in the memory of each microprocessor.

II.D.2.a Function specification The main program establishes three distinct data flow paths for the following modules. In addition, a data structure is created and maintained to be consistent with the abstract notion of data independence. That is, data is passed to the application code and maintained in compliance with the following abstract interpretation.

II.D.2.b Abstract interpretation The DL for the individual data elements consists of the union of the DL associated with each element. The data flow within the application environment emulates that of the hardware configuration. That is, there are three distinct data-flow paths within the application environment, one associated with the three sensor channels in Figure 1. Prior to the call to the initial module, the three DL's are

1. Data Path A — {Chan A}
2. Data Path B — {Chan A, Chan B} and
3. Data Path C — {Chan A, Chan C}.

The interpretation associated with the DL representation is that at this point in the calculation the data corresponding to the data path A calculation is dependent on Chan A. This models the physical system in that the sensor for channel A is connected to processor A. The proof of fault tolerance for the hardware established that, following a fault-tolerant exchange, i.e., before placing congruent data in each channel, the dependencies within an FCR are limited to the components of that FCR.

II.D.3 Signal validation

II.D.3.a Function specification The function of the first module is to validate the signals. Typical signal validation algorithms are based on a statistical hypothesis test, a test that determines whether two signals, emanating from sensors of the same type and monitoring the same parameter, come from the same distribution. For the purposes of this specification, the module will perform a pairwise test for the three combinations of signals available for each pump.

1. Input — two integers, each representing flow in one data path,
2. Output — a Boolean value stating whether the two values are within the same distribution.

II.D.3.b Abstract interpretation This module provides a pairwise comparison for the three combinations of data flow paths. From an abstract function perspective, a comparison is modeled as the union of the DL's associated with each data path. That is,

1. Path A — $\{Chan_A\} \cup \{Chan_A, Chan_B\}$
2. Path B — $\{Chan_A, Chan_B\} \cup \{Chan_A, Chan_C\}$
3. Path C — $\{Chan_A\} \cup \{Chan_A, Chan_C\}$

For example, the interpretation of item 2 is that the DL following a call to the signal validation module with values from Chan_B and Chan_C is {Chan_A, Chan_B, Chan_C}.

II.D.4 Sensor validation

II.D.4.a Function specification The previous module determined whether two parameters were from the same distribution. The function of this module is to validate sensors using the information gained from this determination. Respective to a data path, if the sensor in that path is valid respective to a signal from either of the other paths, then that signal represents valid sensor information. That is, if signal A agrees with either B or C, then sensor A data is said to be valid.

1. Input — Agree(A,B) and Agree(A,C)
2. Output — Is_ok(A)

II.D.4.b Abstract interpretation This module performs a logical *or* of the output from the previous module. The abstract functionality for this module is derived from the effect of this data combination upon the DL abstraction. This function is modeled as the intersection of the DL's associated with the inputs, i.e., $DL_{agree(a,b)} \cap DL_{agree(a,c)}$

The interpretation of this abstraction is that the dependencies of the valid sensor data are only those that are common to the inputs to the module. This models the independence between channels in the data flow specification. If there is an error resulting from a hardware fault in either input data, it affects only one datum. The result is free of faults, as the faulted channel has been masked from the calculation.

II.D.5 Generate trip

II.D.5.a Function specification If a data path contains valid data, then it is used to determine whether flow is adequate. That is, actual flow as determined by each data path is compared to an absolute value. For each path, a determination for appropriate action is made.

The function of this module is to determine a consensus of two data paths for an ultimate decision on action, i.e., whether to trip the reactor or not. A consensus results from the comparison of data from neighboring data paths to an absolute value. A consensus agreement results in setting the output to True.

1. Input — Sensor data and Is_ok, for pairwise combinations of data paths, i.e.,
   a. A and B,
   b. B and C
   c. A and C.
2. Output — Trip determination for data path

II.D.5.b Abstract interpretation The logical construction of this module is the *and* of two data flow paths. This results in the union of their respective DL's.

II.D.6 Vote on trip

II.D.6.a Function specification A vote is taken among the data paths to find a consensus of action to execute.

1. Input — Trip determinations from each data path
2. Output — A consensus for trip action

II.D.6.b Abstract interpretation The logical construction of this module is the *or* of three data flow paths. This results in taking the intersection of all DL's.

A physical interpretation of this abstraction is that this module combined with the generate trip module forms a two-out-of-three vote on the decision to trip. The generate trip module took the consensus of two decisions, i.e., A&B, A&C, and B&C. This was a pure logical consensus in that if A&B were true than path A is true. The result of this module is the

logical *or* of all three data paths. The logical interpretation of these two modules is stated: $(A \wedge B) \vee (A \wedge C) \vee (B \wedge C)$. The abstract interpretation of these modules combined is $(A \cup B) \cap (A \cup C) \cap (B \cup C)$. That a pairwise union of DL's is formed in the generate trip module and the intersection of the three resultant DL's is formed in the vote on trip module.

II.D.7 Conclusion

Following a symbolic execution of the abstract interpretation defined for the software specification, a determination that the dependencies for each FCR holds. That is, the FCR for processor A, with the application program considered, is independent of the other processor FCR's. If the property of independence holds, then the system is tolerant of single failures as shown in [16, 17]

III FAM Specification

III.A Introduction to FAM

This section introduces the first representation of the specification. This representation has been developed at Argonne National Laboratory over the past five years [16].

Figure 4 is a visual representationspecification for the application software. The FAM approach consists of creating a visual model of the system being specified, representing this net in form accepted by an automated theorem prover, and performing symbolic execution using an automated theorem prover. The intent of this section is to describe the visual representation of the specification described above. Details respective to representing the specification in the language of a theorem prover and subsequent symbolic execution are beyond the scope of this paper. The following is an introduction to the constructs used in this description.

Figure 4 depicts a "Flow Net" representation of the subject specification. Flow nets use the basic components of Petri nets but have several additional features that make them more versatile. A Petri net comprises two nodes;

1. Places (circles) and
2. Transitions (horizontal bars).

Nodes are connected by directed arcs and multiple arcs are permitted from one node to another. Another primitive concept for Petri nets are Tokens. Tokens reside in Places and are used to define execution of a Petri net. A *marking* of a Petri net consists of a number of Places being assigned tokens. The *firing* of a Transition occurs when all input places are assigned token(s). In this sense, a Petri net may be used to model state and execution to model system operation. See [19] for further discussion on Petri net theory and modeling.

We have developed a discipline for the annotation of flow nets and have provided extensions to the classic Petri net model in order to define the functional capability that is required for representation of software and hardware. Two of these extensions are of key importance: Tokens are allowed to be symbolic expressions which can contain information representing properties of the system, and Places and Transitions can have a type property associated with them. The firing of a flow net will represent the action of the symbolic function associated with

a Transition operating on the symbolic expressions which are the tokens. For software, Places typically represent the status of the actual program counters; a Token, if present, is the state of the processor at this point; and Transitions represent one or more program statements.

Figure 4 is a flow net description of the specification. A flow net comprises the following entities: 1. Places (circles) 2. Transitions (horizontal bars), and 3. Tokens (sets). Each Place and Transition is identified by a unique name. On Figure 4 these names are placed to the right of the entity: i.e., Ready to Perform SigVal_ab identifies the first-level Place, and Calculate SigVal identifies the second-level Transition. The arrows represent data flow. To the left of each Place is listed an abstract data type that represents the state. This state is based on the representation of an abstract function. The functionality depicted by this figure is dependency list manipulation.

Figure 4 depicts the marking as assigned during execution of the specification being modeled. That is, the tokens assigned to the Ready to Perform SigVal or first level of the net, depict the initial marking for the net. Subsequent firing of the first level transitions (Calculate Sig Val_), move tokens to the next level of Places. When a Transition is fired, Tokens are removed from the input Places and assigned to the output from the Transition. Control flow is modeled by firing of Transitions and data flow is modeled by token assignment.

III.B Details of specification

The first Transition represents the separation of the data flow into three paths. The abstract functionality associated with the first Transition represents the data spreading into three independent data flow paths.

The next level of Places depicts the state of the program prior to execution of the first module and the respective dependencies for data at this state. That is, S(a) is dependent on FCR Pa, represented by {Pa}, S(b) on FCR's Pa and Pb. The interpretation of this representation is that the data in data path B was initially dependent on Pb (the FCR that acquired the data) and that the data currently resides in FCR Pa.

The firing of the second-level Transitions, e.g., Calculate SigVal_ab, represents execution of a module that has the abstract functionality of forming the union of the DL's associated with the input parameters. That is, for data path A, SigVal performs the union of {Pa} and {Pa,Pb}, and the state depicted by the result resides in the subsequent Place, "Ready to Validate Sensor a".

In this fashion, the representation of Figure 4 details the abstract interpretation of the specification.

IV Z Specification

IV.A Introduction to Z

Z is a language based on typed set theory that provides a syntax for expressing mathematical notations. That is, Z provides a language for presenting the mathematical notation that describes in a precise way the properties of a system. The elements of the mathematical notation expressed by Z are:

1. mathematical data types, and

Generic Model for Sensor Validation Software

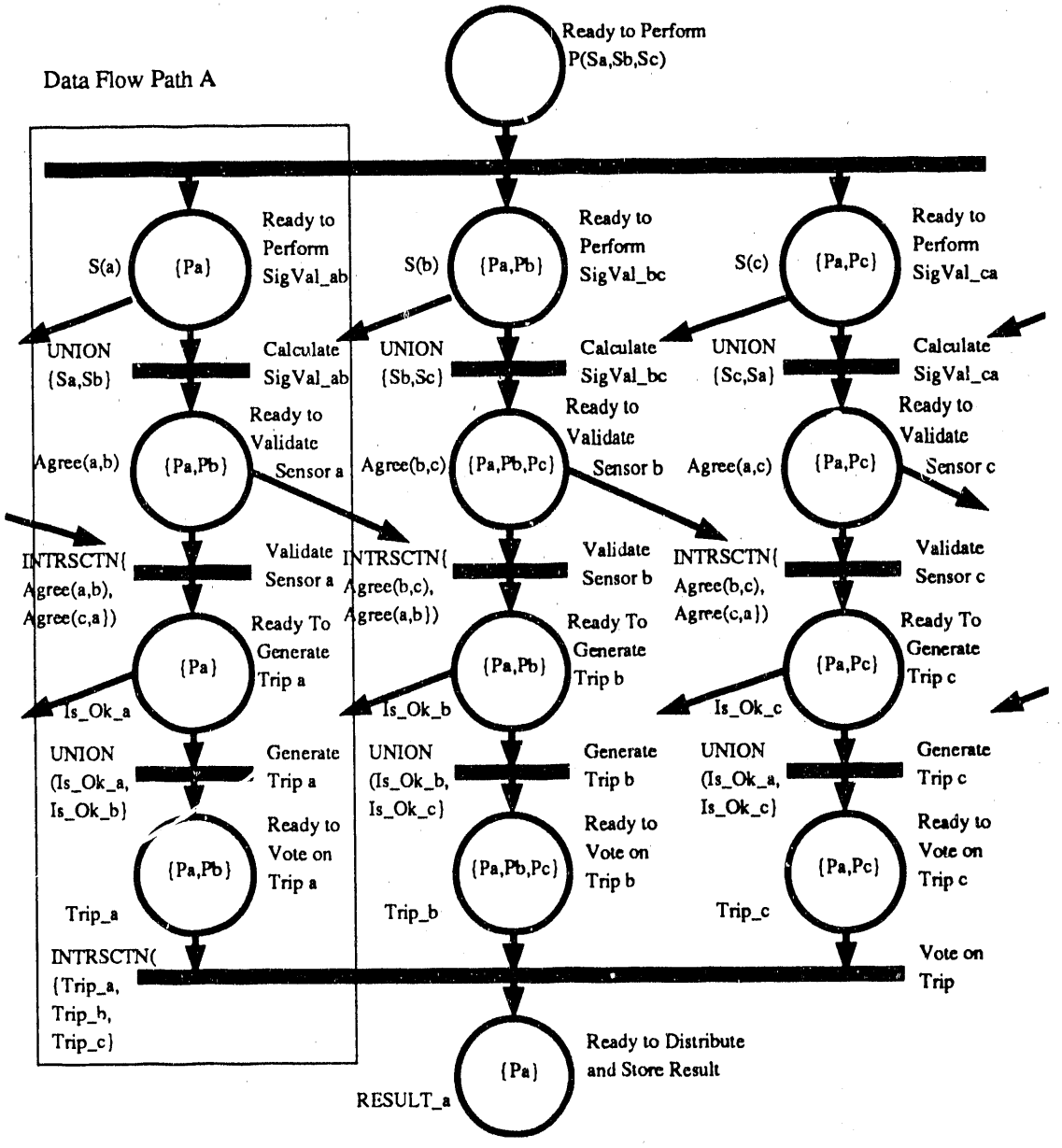

Figure 4 Abstract Specification for the Application Program — FAM

2. schemas.

Mathematical data types are an abstract representation of data in the system and a collection of laws, expressed in predicate logic, describing system properties. Schemas are mathematical expressions that are used to describe aspects of the system, e.g., state, invariant relationships, operations, and functions.

The following consists of a brief introduction to the **Z** representation for the subject case study. This introduction describes the **Z** notation used in subsequent specification and is consists of paraphrasing and extracts from [6, 20].

IV.B Objects and Types

Every mathematical expression in **Z** is given a `type` The idea behind this approach is to provide a way to automatically check that all expressions make sense from a type standpoint. For example, the equation$(a, b) = \{a, b, c\}$ is nonsense in **Z**. However, a specification written without type errors could be inadequate respective to the functionality of the system being specified.

Each **Z** specification begins with a determination of atomic objects, i.e., objects that are essential to the specification but whose internal structure is of no interest. For the example included here, the atomic objects represent abstract data types that require further refinement to form an unambiguous specification. For the purposes of this paper, the atomic objects require no further refinement, since completeness is not the objective.

From the atomic objects, composite objects are constructed. **Z** restricts composition to three composite types: set types, Cartesian product types, and schema types. Type constructors may be applied repeatedly.

IV.B.1 Sets and set types

"Any set of objects of the same type t is itself an object with the `set type` $\mathbb{P}\, t$." [6] Examples: $\{1, 2, 4, 8, 16\}$ has type $\mathbb{P}\,\mathbb{Z}$ and is a set of integers; $\{data : DATA | upperlimit(data) \leq 550\}$ has type $\mathbb{P}$ `DATA` and is the set whose members are exactly those objects of the basic type `DATA` for which the function `upperlimit` has value at most 550. Equality of sets occurs when sets have the same members.

IV.B.2 Tuples and Cartesian product types

"If x and y are two objects of types t and u respectively, then the ordered pair (x, y) is an object with `Cartesian product type` $t \times u$. More generally, if $x_1, \ldots, x_n$ are n objects of types $t_1, \ldots, t_n$ respectively, then the ordered n–tuple $(x_1, \ldots, x_n)$ is an object of type $t_1 \times \cdots \times t_n$."$x_1, \ldots, x_n : t_1, \ldots, t_n \rightarrow (x_1, \ldots, x_n) : t_1 \times \cdots \times t_n$[6]."

The subsequent discussion describes the use of tuples for passing multiple inputs to a function. A specific type used in the Z specification for the case study is a sequence. A sequence is the Cartesian product between the natural numbers and another element, i.e., $\{f : \mathbb{N} \twoheadrightarrow X | dom\ f = 1..\#f\} : \mathbb{N} \times t$, where x is of type t.

IV.B.3 Bindings and schema types

"If p and q are distinct identifiers, and x and y are objects of type t and u respectively, then there is a binding z with the components $z.p$ equal to x and $z.q$ equal to y. This binding is an object with the *schema type* $\langle p_1 : t_1; \ldots; p_n : t_n \rangle$ [6]".

IV.B.4 Relations and functions

The atomic objects that make up the type system for **Z** are sets – $\mathbb{P}$; tuples – $\times$; and bindings $\langle\rangle$ All other objects are modeled by formulating a combination of atomic objects.

"Among the most important mathematical objects are binary relations and functions, and both are modelled in **Z** by their *graphs*. The graph of a binary relation is the set of ordered pairs for which it holds. Mathematical functions are considered as a special kind of relation: those which relate each object on the left to at most one object on the right" [6].

In **Z** a function is a mathematical entity. This contrasts with the "algorithmic" view of functions inherent in program construction. A mathematical function establishes static relations between arguments and results. An algorithmic view regards a function as a method of computation; i.e., the result is computed from the argument.

Consistent with the previous statements about types and automatic type checking, types for functions and relations are identical: that is, if two variables f and g are declared by $f : A \leftrightarrow B; g : A \rightarrow B$ then they have the same type, namely, $\mathbb{P}(A \times B)$, i.e., a set type with objects being Cartesian product types.

IV.B.5 Properties and schemas

Consistent with mathematical notation, the **Z** notation uses *schemas* to describe a specification. A schema groups variable declarations (the signature of the expression) with a list of

Schema

$a : \mathbf{Z}$ $b : \mathbb{P}\mathbf{Z}$	Signature (declaration)
$a \in b$	Property (predicate)

predicates (the properties that hold over the signature). Each schema has a name, e.g., *Schema*. Instantiations of variables that characterize truth of a property over the signature satisfies that property. Propositional logic connectives may be used to combine predicates and to express more complicated properties.

IV.B.6 Decorations

There are three standard *decorations* in the **Z** notation, i.e.,

1. $'$ for labelling the final state of an operation;
2. ? for labelling its inputs; and
3. ! for labeling its outputs.

IV.B.7 Combining schemas

If D is a declaration, P is a predicate and S is a schema, then $\forall D|P \bullet S$ is a schema. The schema S must have as components all the variables introduced by D, and they must have the same types. The signature of the result contains all the components of S except those introduced by D, and they have the same types as in S. The property of the result is derived as follows: for any situation for the signature of the result, consider all its extensions to the signature of S. If every such extension that satisfies both the constraint of D and the predicate P also satisfies the property of S, then the original situation satisfies the property of $\forall D|P \bullet S$. For example: $\forall n : \mathbf{N}|n > 3 \bullet square$, where *square* is described by Figure 5 [6]. . The result is described

square

$y{:}\mathbf{Z}; n : 1..10$
$y = n^2$

Figure 5 Schema – square

by Figure 6.

square'

$y{:}\mathbf{Z}$
$\forall n{:}\ \mathbf{Z}\|n > 3 \bullet$ $\quad z \in 1..10 \wedge y = n^2$

Figure 6 Schema – square'

IV.B.8 Conclusion

This brief introduction to the **Z** notation was provided to facilitate understanding of the following specification. The level of description for the specification allows use of a subset of the rich notation. For example, generic constructions, nested variable scopes, and combinations of schemas were not necessary for the case study. Note that the specification pertains to the abstract interpretation of the specification and that functions are identified but not defined. Definitions would be via schemas that would describe the functions to be algorithmically implemented by the modules.

IV.C Details of specification

Figure 8 depicts the **Z** specification for the case study. This specification provides a state description for the state of the system being controlled, i.e., reactor operation. Most examples of **Z** specifications [6] describe software systems. This specification represents a departure from that approach. The decision to compare specifications resulted in additional effort in that the representation could have been mathematically elegant, yet the meaning more obtuse. The primary difficulty in developing this specification was gaining assurance that the specification was complete. For reasons of clarity, the mathematical representation was chosen for explicit description versus an elegant yet obtuse representation.

This specification will be described by first concentrating on the specification for the module that determines signal agreement (Figure 7). Let DL and AGREE be the basic types.

[DL,AGREE].

The name of this schema is Determine Sensor Agreement. The signature for this schema identifies three entities: 1) a data type that is input (sensor_data?) with a type sequence that is a Cartesian product of the natural numbers and basic type DL; 2) sensors_agree, another sequence that is the Cartesian product of basic types AGREE and DL. The last part of the signature describes a function named SigVal.

The first part of the property description states that the cardinality of sensor_data is three. The intent of this property is to limit the length of the sequences to three elements. The interpretation of the sequences is that each element applies to a data flow path; i.e., the first applies to A, the second to B, and the third to C. The decision to adopt this representation limits the generality of the specification, but facilitates readability.

The second part of the signature states that the input to the SigVal function is data.

The third property states that the sequence "sensors_agree" is a sequence formed as follows for all n & n1 in sensor_data, the first element in each tuple is determined by input n and n1 to function SigVal. The second element is formed by the union of the second element of n with that of n1. and n is less than n1. This results in the sequence a,b, a,c and b,c.

As in the other instances of the specification, this representation lists the functions but does not describe them.

The remainder of the specification is described in Figure 8. The substitution of Ξ for Δ denotes that the reactor state is altered by this specification. For this specification the atomic objects are

[DL,AGREE,VALID,TRIP,RESULT].

Determine Sensor Agreement

$$
\begin{array}{l}
\Xi Reactor\ Operation \\
sensor_data? : seq\langle \mathbf{N} \times DL\rangle \\
sensors_agree : seq\langle AGREE \times DL\rangle \\
SigVal : (\langle \mathbf{N} \times \mathbf{N} \times DL\rangle \times \langle \mathbf{N} \times \mathbf{N} \times DL\rangle) \rightarrow AGREE \\
\hline
\#sensor_data, sensors_agree = 3 \\
sensor_data = \mathrm{dom}\ SigVal \\
sensors_agree = \langle \forall n, n_1 : sensor_data \mid \\
\quad ((\pi_1(sensor_agree(n)) = SigVal(n, n_1)), \\
\quad (\pi_2(sensors_agree(n)) = (\pi_2(sensor_data(n)) \cup \\
\quad\ \pi_2(sensor_data(n_1))) \wedge (n < n_1)\}
\end{array}
$$

Figure 7 Z Specification for Determine Sensor Agreement

Generate a Reactor Trip Signal on Detection of Low Primary Coolant Flow[6]

$\Delta Reactor\ Operation$

$sensor_data? : seq\langle \mathbf{N} \times DL\rangle$

$sensors_agree : seq\langle AGREE \times DL\rangle$

$is_ok : seq\langle VALID \times DL\rangle$

$trip : seq\langle TRIP \times DL\rangle$

$SigVal : (\langle \mathbf{N} \times \mathbf{N} \times DL\rangle \times \langle \mathbf{N} \times \mathbf{N} \times DL\rangle) \rightarrow AGREE$

$Validate : (\langle \mathbf{N} \times \mathbf{N} \times DL\rangle \times \langle \mathbf{N} \times \mathbf{N} \times DL\rangle) \rightarrow VALID$

$Gen_Trip : \langle \mathbf{N} \times \mathbf{N} \times DL\rangle \rightarrow TRIP$

$Vote : \langle \mathbf{N} \times \mathbf{N} \times DL\rangle \rightarrow RESULT$

$\#sensor_data, sensors_agree, is_ok, trip = 3$

$sensor_data = \mathrm{dom}\ SigVal$

$sensors_agree = \mathrm{dom}\ Valid$

$is_ok = \mathrm{dom}\ True$

$trip = \mathrm{dom}\ Vote$

$$
\begin{aligned}
sensors_agree = \langle \forall n, n_1 : sensor_data \mid & \\
& ((\pi_1(sensor_agree(n)) = SigVal(n, n_1)), \\
& (\pi_2(sensors_agree(n)) = (\pi_2(sensor_data(n)) \cup \\
& \quad \pi_2(sensor_data(n_1))) \wedge (n < n_1)\}
\end{aligned}
$$

$$
\begin{aligned}
is_ok = \langle \forall n : sensors_agree \mid & \\
& ((\pi_1(is_ok(n)) = Validate(n, n_1)) \wedge (n < n_1) \wedge (n, n_1) \neq (n_1, n)), \\
& ((\pi_2(is_ok(1)) = (\pi_2(sensors_agree(1)) \cap \\
& \quad \pi_2(sensors_agree(3)))) \wedge \\
& (\pi_2(is_ok(2)) = (\pi_2(sensors_agree(1)) \cap \\
& \quad \pi_2(sensors_agree(2)))) \wedge \\
& (\pi_2(is_ok(3)) = (\pi_2(sensors_agree(2)) \cap \\
& \quad \pi_2(sensors_agree(3))))))\rangle
\end{aligned}
$$

$$
\begin{aligned}
trip = \langle \forall n : is_ok; sensor_data \mid & \\
& \pi_1(trip(n)) = (((\pi_1(sensor_data(n)) \leq 88\% full flow) \wedge \\
& Gen_Trip(\pi_1(is_ok(n))), \\
& \pi_2(trip(n)) = \pi_2(is_ok(n))\rangle
\end{aligned}
$$

$$
\begin{aligned}
\pi_1(result) &= Vote(trip) \\
\pi_2(result) &= \langle \langle \pi_2(trip(1)) \cup \pi_2(trip(2))\rangle \cap \\
& \quad \langle \pi_2(trip(1)) \cup \pi_2(trip(3))\rangle \cap \\
& \quad \langle \pi_2(trip(2)) \cup \pi_2(trip(3))\rangle\rangle
\end{aligned}
$$

Figure 8 Z Specification for Generic Algorithm

[6] The intent of this specification is to assure that the essential property of independence holds. This property is an abstraction of system-wide fault-tolerance.

V STATEMATE®[7] Specification

V.A Introduction to STATEMATE

V.A.1 Background

STATEMATE [21] is comprised of a set of tools that provide an environment for modeling systems. This environment uses a graphic input language called *statecharts* [8, 9]. Statecharts are based on an extension of state-transition diagrams and are used as a graphical representation of complex systems. Using the STATEMATE environment, this representation may be used to design and analyze models of a system. These models are based on a set-theoretic formalism, the elements that were essential to the subject specification are described below. This description uses the statechart representation. STATEMATE was unavailable for this case study.

V.A.2 *Higraphs*

Roundtangles enclose *blobs*. A blob denotes a set, with the nesting of curves denoting set inclusion, not set membership. Figure 9 shows the requirement that every set of interest be represented by a unique blob. That is, the two large intersecting blobs in Figure 9 are labled A and D, the intersection $A \cap D$ is labeled C, and the difference $A - D$ is called B. The only identifiable sets are the *atomic* sets, i.e., those represented by the blobs residing at the bottom levels of the diagram (containing no wholly enclosed blobs within). The atomic blobs in Figure 9 are B, C,E, G, H, I, K, L, M, N, O, Q, S and T. Note that $T \cap R$, and $T - R$ have no meaning.

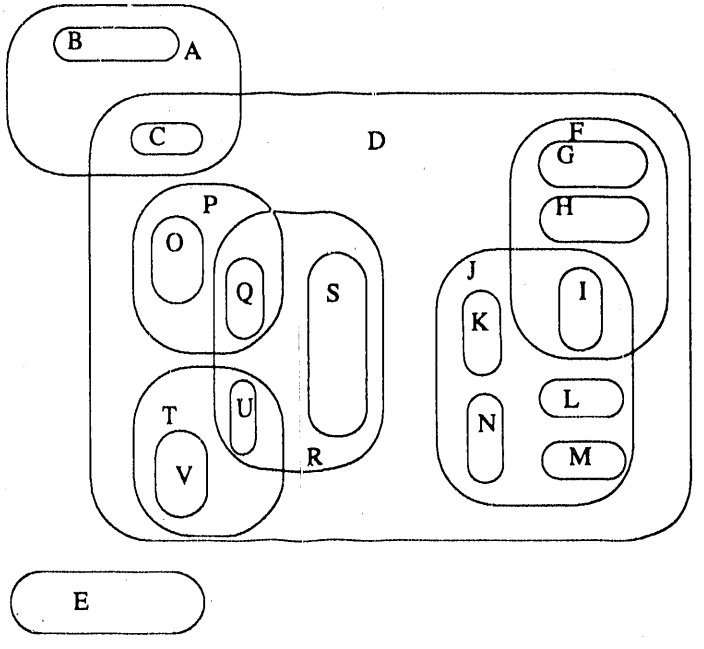

Figure 9 Simple blobs and unique contours describe all identifiable sets

[7] STATEMATE is a registered trademark of i-Logix, Inc.

V.A.3 *Hyperedges*

Hyperedges are edges that are attached to the contour of *any* blobs. Edges may be directed or undirected, labeled or unlabeled, of one type or of several, etc. Within this context *arity* refers to the number of blobs being connected.

V.A.4 Discussion

Consistent with the previous two examples, the STATEMATE specification represents the abstract interpretation of the case study. In this example, set manipulation is the exclusive functionality being depicted. STATEMATE provides a natural representation for this specification. As in the preceding examples, the concrete functionality is not represented. In addition, the function between sets has been excluded. Annotation of the abstract interpretation would naturally be ascribed to the arc between sets. This has also been omitted since the diagrams became complex and difficult to read. It was felt that inserting the function names was a simple exercise based on the previous specification descriptions.

V.B Details of specification

The representation chosen is to use the atomic DL as the basic type (Sa, Sb, Sc in Figure 10). That is, Sa depicts the initial dependencies at the context switch to this module, i.e., {Pa}.

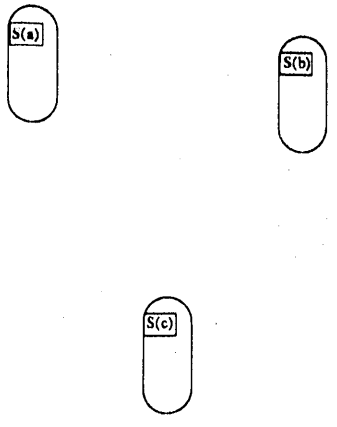

Figure 10 STATEMATE specification — Level 1 — DL's associated with sensors

Figure 11 depicts the union of sets S(a) and S(b) by the blob labeled Agree(a,b). This represents the abstract interpretation of the module SigVal.

Figure 12 depicts the intersection of the Agree sets; i.e., the blob IsOk(a) is the intersection of sets Agree(a,c) and Agree(a,b).

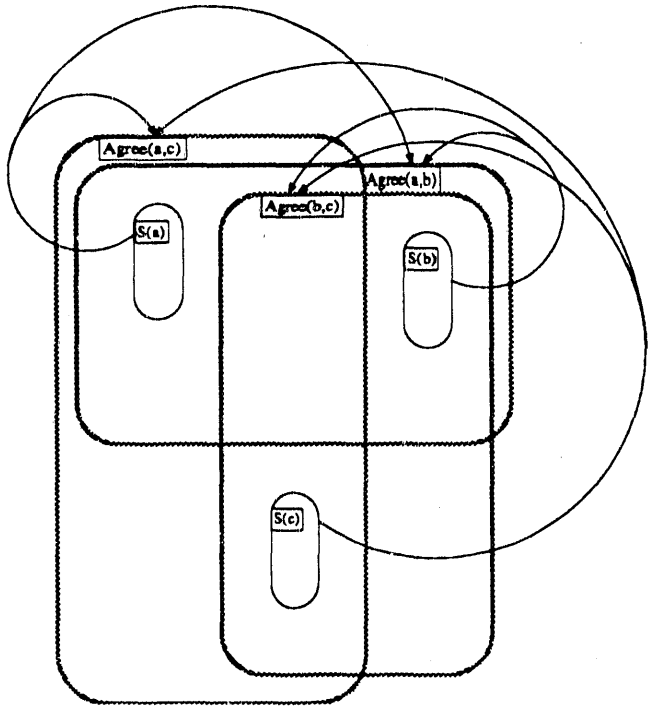

Figure 11 STATEMATE specification — Level 2

Figure 13 represents the union of sets to emulate the functionality of module Trip; i.e., S(a) unioned with IsOk(A) forms Trip(a).

Figure 14 depicts the union of Trip sets; i.e., Trip(a,b) is the union of Trip(a) and Trip(b).

The Result is determined by the intersection of three sets.

VI Summary and Conclusion

The logical basis for this paper was rooted in the observation that computers were increasingly being used for the control of life-critical systems. From this observation, an argument was given for the adoption of a methodology for system design that was based in formal system specification. A literature review defined an area of research called *Formal Methods*. A subset of FM's were discovered to be applicable to specification development, and representative methods from this subset were selected to conduct a case study.

The case study selected was the specification of a control subsystem for an operating nuclear reactor. For the purposes of this case study, an abstract interpretation for the functionality of the software being specified was developed. This abstract interpretation is consistent with previous abstractions applied to the analysis of the fault-tolerant processor on which this system operates.

The reader should note that the selection of abstract interpretations and their representation are a subjective exercise. The following comparison information should be viewed as being biased by the author's view of conceiving specifications and the relative match between this model and those underlying the three specifications presented. Stated differently, creating a

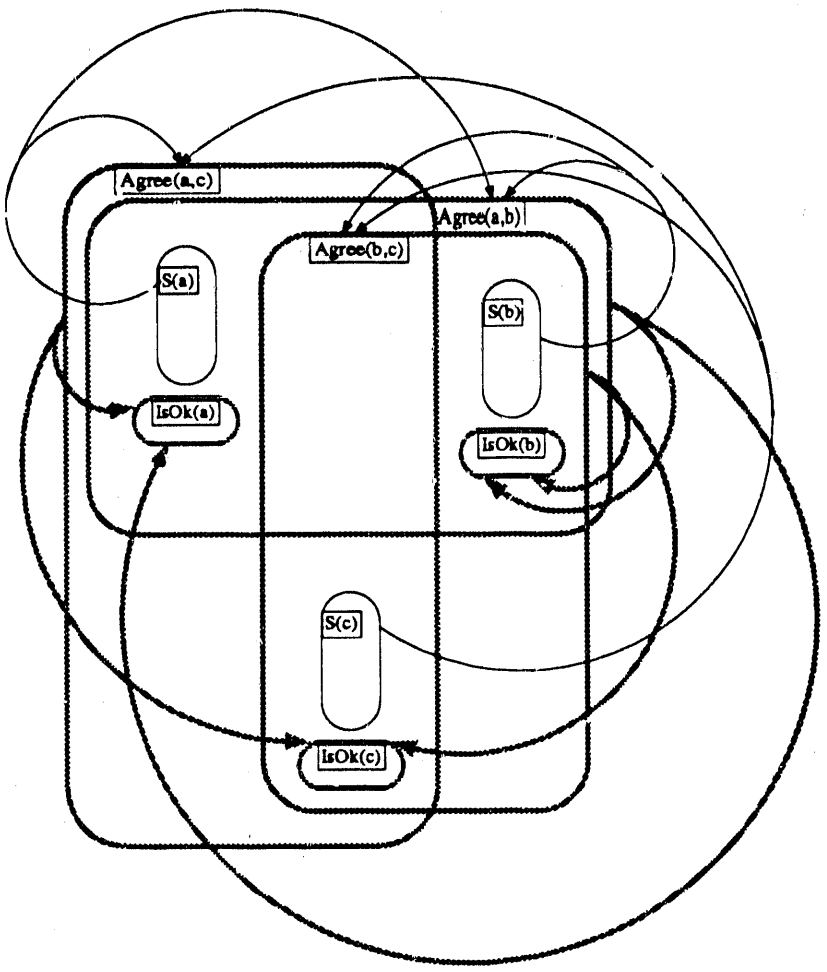

Figure 12 STATEMATE specification — Level 3

specification at this level of abstraction may be more of an art than a formal exercise. The expressive nature of one specification notation differs from others and may be mismatched with that of the specifier. Recognition of these limitations suggests that a specification system may be adequate for one designers conceptualization of a total system, but may be inadequate for another's.

The abstract interpretation approach to high-level specifications provided sufficient insight into the functionality of the design that work is currently pursuing the completion of a formal specification of functionality. The pattern is that of the approach to abstract data structures. That is, the abstract interpretation defines the data and control structure for the application software. Following a formal understanding of this structure, the functionality is understood and subject to formalization.

The intent of the case study was to compare the representations of three different FM's to determine their respective merits. In the introduction, it was postulated that completeness and unambiguity were properties of specifications used for correctness proofs. Construction of designs that are correct with respect to essential properties should result in a hierarchical structure that is complete and unambiguous. Therefore, the first comparison of the three FM's was on their respective merit in producing complete and unambiguous specifications. However,

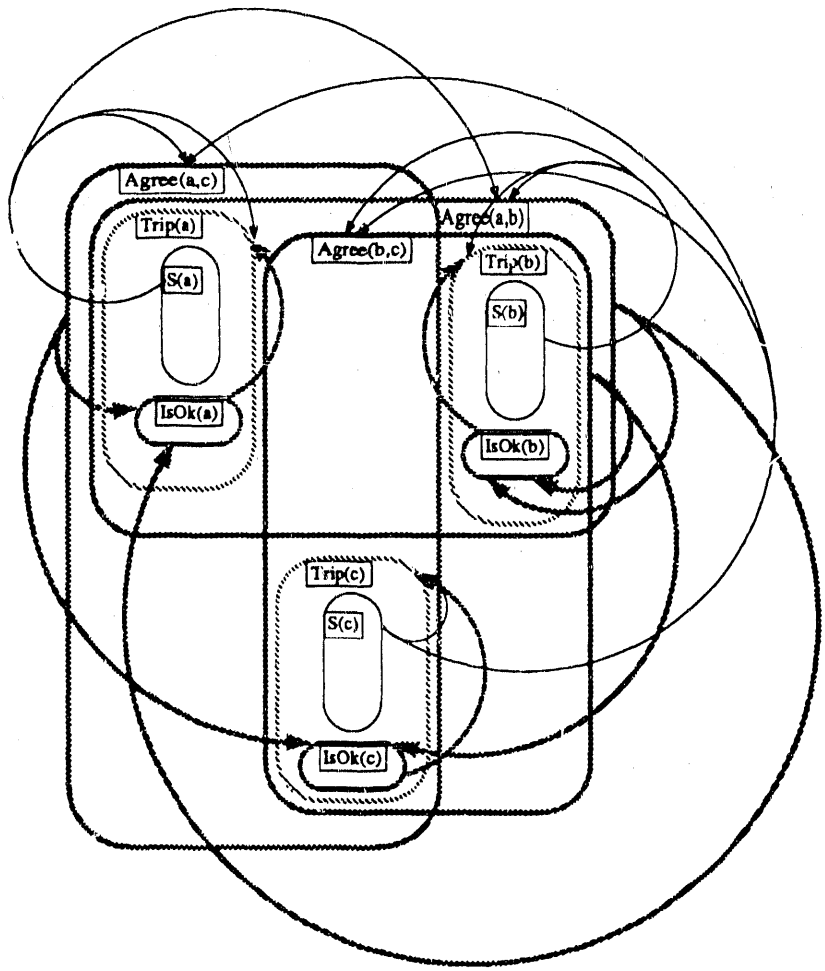

Figure 13 STATEMATE specification — Level 4

any comparison based on completeness was inconclusive. Since the abstract interpretation was based in set theory, a precise statement was possible in each representation.

A comparison of three specifications with the figure of merit being ambiguity is difficult, and is often the subject of critiques and theses. This difficulty is further compounded by the fact that the creator of all three specifications and the reviewer are the same person. Therefore, the comparison will be based on the experience gained from creating three specifications for a single design. That is, the comparison will be a subjective conclusion drawn from the experience of transforming an abstract interpretation of a design into three diverse representations. Figure 1 summarizes the conclusions drawn from this experience.

The invention of an abstract interpretation raised the level of abstraction to an intuitive level that was more easily expressed by the visual languages. However, the decision to use sequences to associate channels with objects in a sequence, complicated the**Z** specification. The complication was created by the necessity of using decomposition functions to select sequence elements. Though the choice of representation was unfortunate, it also indicates that the representation for a **Z** specification is less intuitive.

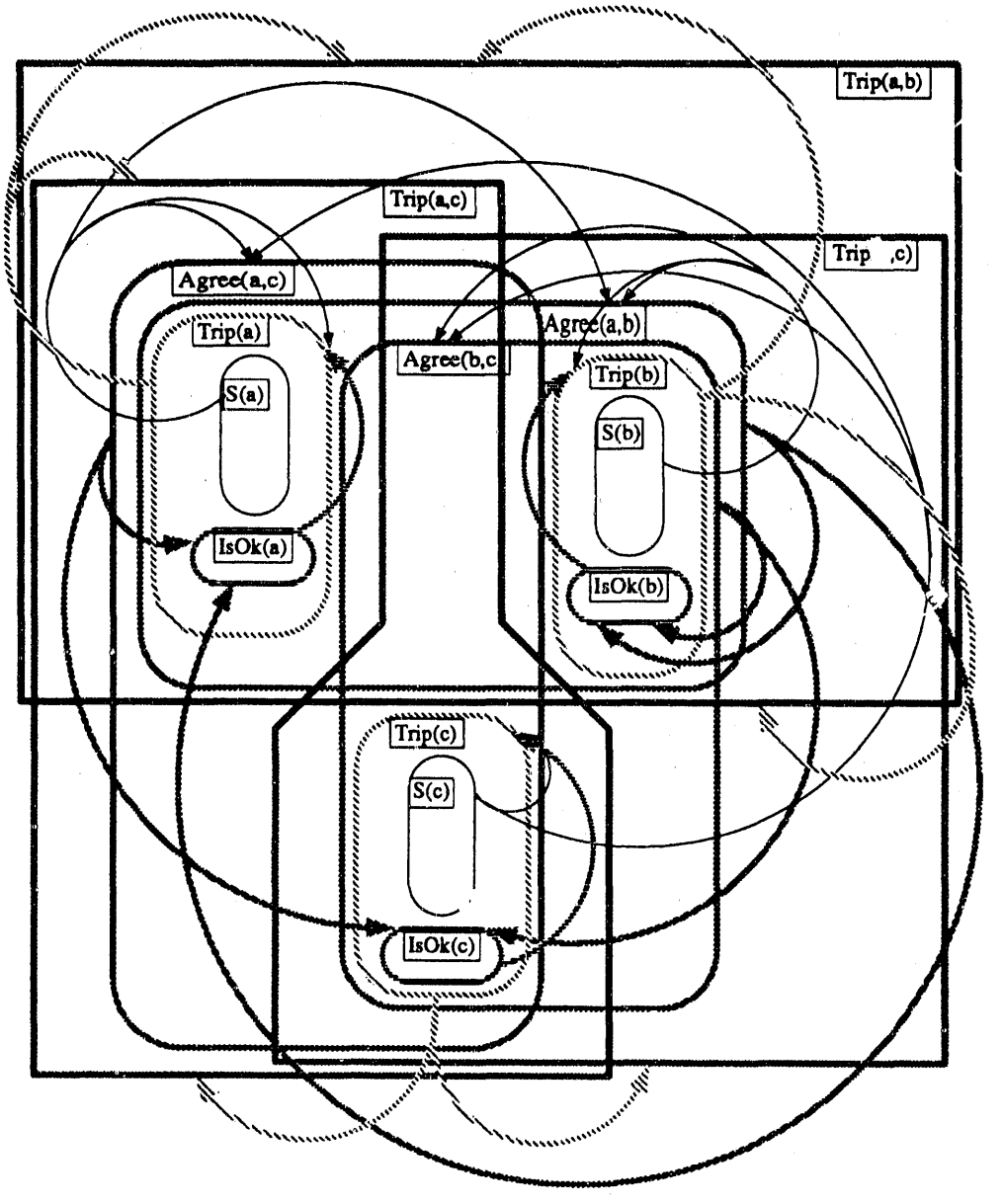

Figure 14 STATEMATE specification — Level 5

Table 1 Summary of Comparison

Formal Method	Comments	Observations
FAM	A combination of visual and textual representations (Though only the visual is presented as part of the case study)	Explicitly depicts data and control flow.

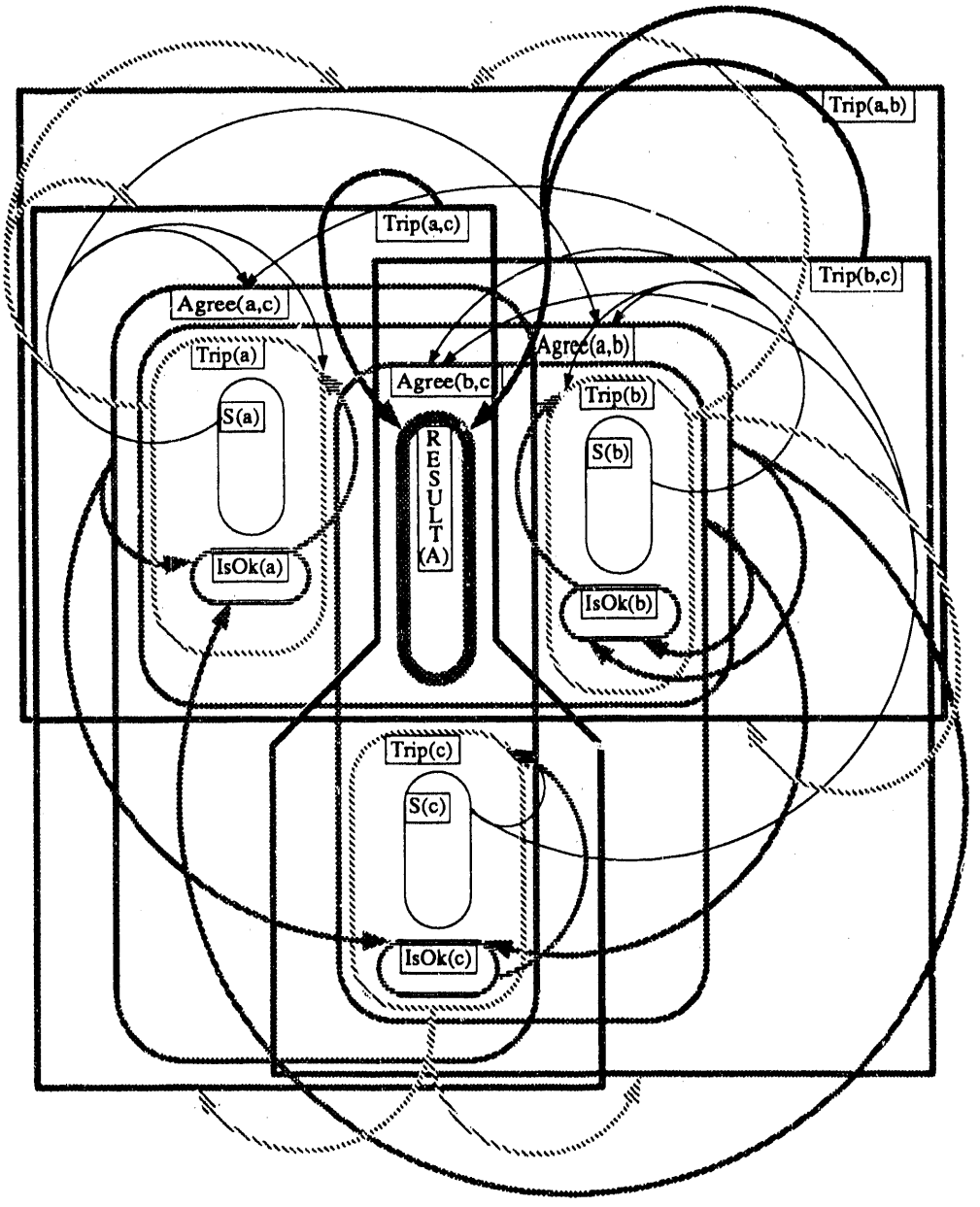

Figure 15 STATEMATE specification — Complete

Table 1 (Continued) Summary of Comparison

Formal Method	Comments	Observations
		Flow nets are executable when represented in a clausal form and submitted to an automated theorem prover.

Table 1 (Continued) Summary of Comparison

Formal Method	Comments	Observations
Z	Mathematical Notation Elegant and precise	Difficult to confirm compelteness, i.e. Choice between concise elegance and detailed description
	Originally conceived for specification of software	A difficult notation to represent hardware specifications, especially timing and control.
STATEMATE®	Visual representation is very intuitive, especially for the abstract interpretation of this case study	Functionality for the abstract interpretation very direct.
		Higraph become very complex very easily. Extensions to the formalism are required to simplify complex specifications. Flow of control provides a further complication

The conclusion of this case study is to identify properties with which to judge the relative merits of a representation to be used in the specification of a total system. This list of properties is based on a philosophy that the creation of a specification is cumulative effort of a number of individuals. Each individual has a unique concept of the system being specified and each concept is based on the expertise of that individual. A superior specification is one that captures the essence of each discipline concerned with the correct operation of the system. It is postulated that the single most essential property is the ability of this document to convey information between the group of specifiers to assure that what is being specified is consistent with each expert view. Once this consensus is complete at a high level of abstraction, then decomposition to a level of specification where ambiguity and completeness may be measured is possible.

Acknowledgments

We are grateful to Gail Pieper for multiple reviews and critical comment. We acknowledge James Boyle for providing poignant advice, suggestions, and assistance.

Bibliography

[1] J. M. Wing, "A specifier's introduction to formal methods," *Computer*, vol. 23, pp. 181–197, September 1990.

[2] J. McLean, "A formal method for the abstract specification of software," *Journal of the Association for Computing Machinery*, vol. 31, pp. 600–627, July 1984.

[3] A. W. Bartussek and D. L. Parnas, "Using traces to write abstract specifications for software modules," Technical Report UNC Rep. TR 77-012, Univ. North Carolina, Chapel Hill, N.C., 1977.

[4] D. L. Parnas, D. G. Smith, and T. Pearce, "Making formal software documentation more practical - a progress report," Technical Report 88-236, Queen's University at Kingston, Kingston, Ontario, Canada K7L 3N6, November 1988.

[5] D. L. Parnas, J. van Schouwen, and S. Po Kwan, "Evaluation standards for safety critical software," Technical Report 88-220, Queen's University at Kingston, Kingston, Ontario, Canada K7L 3N6, May 1988.

[6] J. M. Spivey, *The Z Notation*. New York: Prentice-Hall, 1989.

[7] R. W. Selby, V. R. Basili, and F. T. Baker, "Cleanroom software development: An empirical evaluation," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 9, pp. 1027–1037, 1987.

[8] D. Harel, "Statecharts: A visual formalism for complex systems," in *Science of Computer Programming 8*, pp. 231–274, North-Holland: Elsevier Science Publishers B.V., 1987.

[9] D. Harel, "On visual formalisms," *Communications of the ACM*, vol. 31, pp. 514–530, May 1988.

[10] W. R. Bevier, W. A. Hunt, Jr., J. S. Moore, and W. D. Young, "An approach to systems verification," Technical Report CLI-41, Computational Logic, Inc., Austin, Texas 78703, April 1989.

[11] R. S. Boyer and J. S. Moore, *A Computational Logic Handbook*. San Diego, Calif.: Academic Press, Inc., 1988.

[12] A. Cohn, "The notion of proof in hardware verification," *Journal of Automated Reasoning*, vol. 5, pp. 127–139, June 1989.

[13] G. H. Chisholm, B. T. Smith, and A. S. Wojcik, "Formal analysis of software in a fault-tolerant environment," Preprint MCS-P111-1189, Argonne National Laboratory, Argonne, Ill, 1989.

[14] J. H. Lala, "A Byzantine resilient fault tolerant computer for nuclear power plant applications," in *Proceedings of the International Symposium on Fault-Tolerant Computing Systems*, (Vienna, Austria), pp. 338–342, July 1984.

[15] G. H. Chisholm, J. Kljaich, B. T. Smith, and A. S. Wojcik, "An approach to the verification of a fault-tolerant, computer-based reactor safety system: A case study using automated reasoning," Interim Report NP-4924, Electric Power Research Institute, Palo Alto, Calif., January 1987. Research Project 2686-1.

[16]J. Kljaich, B. T. Smith, and A. S. Wojcik, "Formal verification of fault tolerance using theorem-proving techniques," *IEEE Transactions on Computers*, vol. 38, pp. 366–376, March 1989.

[17]J. Kljaich, B. T. Smith, and A. S. Wojcik, "Verification of fault tolerance using dependancy lists," in *Proceedings of 1988 International Conference on Advanced Science and Technology*, (Chicago, Ill.), February 1988.

[18]A. Bloss, P. Hudak, and J. Young, "An optimising compiler for a modern functional language," *The Computer Journal*, vol. 32, no. 2, pp. 152–161, 1989.

[19]J. L. Peterson, *Petri Net Theory and The Modeling of Systems*. New Jersey: Prentice-Hall, Inc., 1981.

[20]D. C. Ince, *An Introduction to Discrete Mathematics and Formal System Specification*. New York: Oxford University Press, 1st ed., 1988.

[21]D. Harel, H. Lachover, A. Naamad, A. Pneueli, M. Politi, R. Sherman, A. Shtul-Trauring, and M. Trakhtenbrot, "Statemate: A working environment for the development of complex reactive systems," *IEEE Transactions on Software Engineering*, vol. 16, pp. 403–414, April 1990.

END

DATE FILMED

02 / 14 / 91