CONF-8610218--1

Toward Formal Analysis of Ultra-Reliable Computers -- A Total Systems Approach*

CONF-8610218--1

DE87 004817

G. H. Chisholm
J. Kljaich
B. T. Smith
A. S. Wojcik

ARGONNE NATIONAL LABORATORY
Experimental Breeder Reactor - II Division and
Mathematics and Computer Science Division
9700 S. Cass Avenue
Argonne, Illinois 60439
and
Department of Computer Science
Michigan State University
East Lansing, Michigan 48824

Abstract

This paper describes the application of modeling and analysis techniques to software that is designed to execute on four channel version of the the Charles Stark Draper Laboratory (CSDL) Fault-Tolerant Processor, referred to as the Draper FTP. The software performs sensor validation of four independent measures (signals) from the primary pumps of the Experimental Breeder Reactor-II operated by Argonne National Laboratory-West, and from the validated signals formulates a flow trip signal for the reactor safety system.

The work reported here is part of a hardware/software modeling and analysis project. In an earlier paper (2), we have described the application, hierarchical modeling technique based upon Petri Nets (7), and the formal analysis techniques based upon the automated reasoning software ITP/LMA (5). In the earlier paper, we demonstrated the fault-tolerance of the FTP's data exchange instructions to failures in the hardware. In this paper, we demonstrate that the same modeling and analysis techniques apply to proving the fault-tolerance of the software to failures in the hardware, including the sensors, provided the validation algorithms have a certain generic structure. In addition, this approach has provided insight into formal software specification as well as into the generation of test vectors for software. The combination of a formalized specification and a potential formal derivation of test vectors provides continuity between specification, design analysis and testing.

Emphasis is placed upon the hierarchical modeling capabilities of Petri nets. In particular, we have developed an abstraction of the data-flow in terms of the specification of a generic application program for the Draper FTP. Using this program and the data-flow abstraction, we prove the fault-tolerance of the application program to hardware and sensor failures. Finally, based upon a more etailed specification of the sensor validation and flow trip software, we demonstrate that this program satisfies the sufficient conditions developed for the generic program to claim fault-tolerance.

Introduction

This paper is a report of the software verification aspect of a project whose goal is to explore the feasibility of automating the verification process for computer-based safety systems. The intent of the project is to demonstrate that both the software and hardware that comprise the system meet specified availability and reliability criteria, that is, total design analysis. The approach to automation is based upon the use of Automated Reasoning Software developed at Argonne National Laboratory (5). This approach is herein referred to as formal analysis and is based on previous work on the formal verification of digital hardware designs (4). Formal analysis represents a rigorous evaluation which is appropriate for system acceptance in critical applications, such as a Reactor Safety System (RSS). The demonstration of the feasibility of applying formal analysis is the application of these techniques to a case study.

The case study is based on the Reactor Safety System (RSS) for the Experimental Breeder Reactor-II (EBR-II). This is a system where high reliability and availability are tantamount to safety. The conceptual design for this case study incorporates a Fault-Tolerant Processor (FTP) for

*This work was supported in part by the Reactor Core Technology, High Technology Development and Application Program (AF-30-50) and by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under Contract W-31-109-ENG-38, by the National Science Foundation, Grant No. DCR83-17524, and by the Electric Power Research Institute under contract number RP 2686-1.

the computer environment. An FTP is a computer which has the ability to produce correct results even in the presence of any single fault. This Technology was selected as it provides a computer-based equivalent to the traditional analog based RSSs. This provides a more conservative design constraint than that imposed by the IEEE Standard, Criteria For Protection Systems For Nuclear Power Generating Stations (ANSI N42.7-1972).

The Project

Since 1983, the EBR-II Division of Argonne National Laboratory has established the goal of developing and demonstrating a computer-based control system with sufficient availability and reliability for use in reactor safety systems. This goal has been incorporated into a project entitled the Full Authority, Fault-Tolerant, Reactor Control System, FAFTRCS. Significant effort toward attaining the ultimate goal has focussed on investigating available technologies and in synthesizing various disciplines. For example, after studying the availability of fault-tolerant computer systems (8), a decision was made to contract with the Charles Stark Draper Laboratory, CSDL, for the design and fabrication of such a system. The CSDL Fault-Tolerant Processor, FTP, forms the basis for the reactor safety system.

Although guided by this case study, the computer-based solution proposed by the FAFTRCS project poses numerous areas for research. Indeed, if digital systems are to be incorporated into commercial reactors, one must identify the relevant issues concerning verification and validation of computer-based reactor shutdown systems so that they adequately meet the requirements for licensing. This paper addresses the issue that the software for each application must be demonstrated to work reliability in the presence of hardware fault.

One must be cautioned against viewing the results presented in this report as being complete in scope. The goal of the project is to demonstrate the feasibility of the approach and to identify those areas that need further research and development to fully validate the total system. This report thus represents the next step, after (2), in developing a verification methodology and demonstrating its viability on an essential and non-trivial application.

The Case Study

The essence of this case study is to devise a methodology which leads to the qualification of the total system. Thus, the study involves complete analysis of design for meeting the requirements of ultra-high reliability and availability. This section describes the guidelines which govern qualification, the target system, and the design concepts for a subsystem of the target system.

Experimental Breeder Reactor No. II (EBR-II) is a DOE-owned and ANL-operated research reactor. In accordance with DOE guidelines, a Reactor Shutdown System (RSS) is provided to ensure safe reactor shutdown or to mitigate the consequences of postulated accidents. The RSS is designed in accordance with DOE standard RDT C-16-1T. The intent of this standard is to provide guidance during the design and analysis of the RSS to ensure adequate system reliability.

The present EBR-II RSS is briefly described in (2). In summary a subsystem provides protection against plant excursions where the power-to-flow ratio may exceed a predetermined limit. This subsystem will generate a reactor trip signal should this limit be exceeded. The underlying strategy for protection of EBR-II is that the power-to-flow ratio be maintained at or below a predetermined level.

The impetus for this project lies in the continuing loss of the flow metering capability at EBR-II. Since initiating operation, seven out of the original non-replaceable ten flowmeters have failed. To continue adequate protection against loss of flow transients, the FAFTRCS Project is developing a flow protection circuit which will derive a measure of flow from replaceable sensors.

The system being developed by the FAFTRCS Project will replace the two flow protection subchannels of the present RSS with a computer-based system that is functionally equivalent and satisfies the design constraints of reliability and availability. Figure 1.1 depicts the current conceptualization of this subsystem, which is further described in (2).

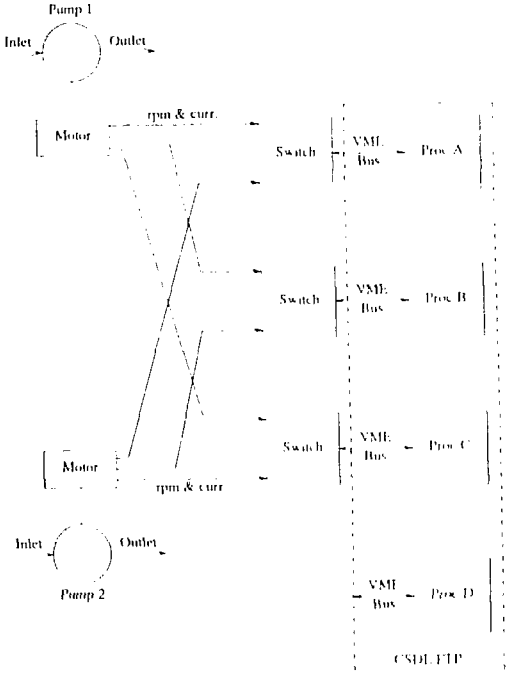

Figure 1.1. Overview of the FAFTRCS Conceptual Design

Replicated sensors are installed on both EBR-II primary coolant pumps to provide greater reliability and availability. After reading in the data from the redundant sensors, software is executed to validate the sensor data and then to infer whether flow is within the accepted guidelines (see Section 4). If not, a reactor trip is generated. The activation of a flow trip by the fault-tolerant processor is based upon a 2-out-of-4 relay logic vote.

As mentioned in the introduction, this report discusses the results of investigations into the feasibility of applying formal analysis techniques to the problem of qualifying computers for use in RSSs. The case study is planned to culminate with the approval for installation of the FTP-based flow protection subsystem into the RSS at EBR-II. This approval is granted by a group of reactor safety experts who are convened to review the documentation prepared to demonstrate compliance with the standards that were previously mentioned.

The Draper Fault-Tolerant Processor (FTP) - Overview

The Draper Fault-Tolerant Processor consists of four identical nodes (each with a Motorola 68000 processor, memory, input/output ports, communicator, interstage register, clock, and busses) connected together through their communicators and interstage registers in an identical manner (11). The nodes are composed of off-the-shelf components and are identical, except for identification 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 every component in all nodes 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 point of view 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 system is designed to remain in synchrony, it is subject to failures which potentially invalidate this design claim. To tolerate such failures, the operating system periodically checks 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. Checks are made by distributing the values of data register, status flags, error indicators, and certain memory values on a regular basis through the communicators and interstage registers. These checks are performed in the processor idle times between execution of the application. As 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.) By this mechanism, a faulted node is detected. The operating system in conjunction with the hardware will reconfigure the FTP system so that the faulted node is masked out and replaced by the fourth spare node.

On the average, this continuing process of checking that the nodes are performing the same operation on the same data occupies at most 10% of the available CPU power. Also, during this time, the fourth (spare) processor is participating in all of the operations in synchrony with the three active nodes. Thus, at any given time, it is known whether or not it is operational and has the same data as the other nodes. Provided the spare is fault-free, it is ready to assume the role of any faulted node that may be discovered among the active nodes. (In the current design, the automatic reconfiguration of the FTP must be accompanied by a manual intervention that connects the sensors of the faulted node to the spare node.)

Synchrony of the node clocks is maintained at the hardware level by a special fault-tolerant clock. Specialized hardware steals cycles, depending on whether a clock signal derived from each node's clock lags, is the same as, or is ahead of the the fault-tolerant clock. This clock check is performed by the hardware using a distribution network similar to that for the data network mentioned above, but reserved for clock data only.

The Softeware For The Draper FTP

The software consists of the operating system and the application software.

The operating system initializes the nodes and configures the processors so that at least three are in synchrony. Once this is accomplished, a dispatcher is started that cyclically executes the application software and other software that ensures the system is tolerant to hardware failures. The software that ensures this tolerance is fast fault-detection, isolation, and reconfiguration software (FDIR), timer check software, and certain hardware checks performed as background tasks.

The application software reads each available plant sensor (twelve for FAFTRCS flow trip computation, four independent measures of three redundant sensors) once and determines the state of the flow for the reactor primary system. If an anomaly in the flow is detected, the application software in each node generates a trip signal which is interpreted by the reactor shutdown system.

The computation time required to determine the state of the plant is small compared to the required time interval between successive readings from the plant sensors. In the slack time, the dispatcher initiates certain tasks that check for and maintain the proper operation of the FTP. These tasks include a timer assessment procedure and a portion of a series of background tasks that check for latent faults in the hardware as well as analyze the results (syndrome bits) of the voting process.

The flow of control in the specific application software is as follows. The first time the application software is executed, initialization of local variables is performed. Once completed, the software initiates, reads, and stores the sensor

data from its own node. Next, the software distributes the data from each node to the others in such a manner that: (1) the data in each processor is the same, if there are no hardware faults, or (2) the data in a majority of the processors is the same (but not necessarily correct), if there are hardware faults. Signal validation tests (sequential probability ratio tests) on the twelve sensor signals are performed, six (three current and three rpm) for each pump. The results for each of these tests from each node are distributed to the other nodes, compared, and distributed back to each node. Again, the data returned to each processor is assured by the hardware to be the same in a majority of the processors but not necessarily correct. However, it should be noted that the program is so structured that a claim can be made that the results are correct even in the presense of one fault. Each node then indicates a flow trip signal, if appropriate, to the reactor shutdown system.

The Draper FTP Hardware

In Figure 1.2, a simplified diagram of the data flow side of the Draper FTP is shown (9-11). Fundamentally, the figure shows the interconnection of four nodes, which will be referred to as Node-A, Node-B, Node-C, and Node-D, each node comprised of

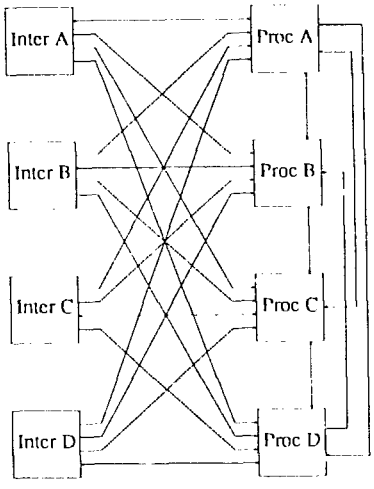

Figure 1.2. Simplified Representation of the Data Flow in the Quadded Draper FTP

a cluster of units, including a processor unit which is a Motorola 68000. As an example, Node-A consists of the following units: an interstage register, Inter-a; a communicator, Com.-a; Versa-Bus; the processor, Proc-a; Memory Mem-a; I/O-Port; VME-Bus; two rpm sensors. The function of each component is discussed in (2).

Simplified Representation of the Data Flow in the Quadded Draper FTP Figure 1.2 is a simplification of Figure 1.2 in 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.) Not described in this figure are the hardware voting circuits that are contained within each communicator. The quadded voting structure contributes to the fault-tolerance of the total system.

The focus of the design of the FTP is its ability to provide congruent data to a majority of the processors. The meaning of congruent data is the following: 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. Notice that congruent does not mean correct. In Figure 1.2, 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. 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 same instructions in lock step synchrony with one another. When a data value is computed by each of the processors, each distributes its value to the others via the data distribution network. 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.

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 fault-containment region malfunctions, at most a single error is generated out of the region. Understand that this 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.

The Interface Between The Draper FTP Hardware and Software

The interface between the hardware and software has been partitioned into two levels:

A level needed to verify claims of fault-tolerance of the total system (see Figure 2.1);

A level to verify claims about the functionality which supports the first level.

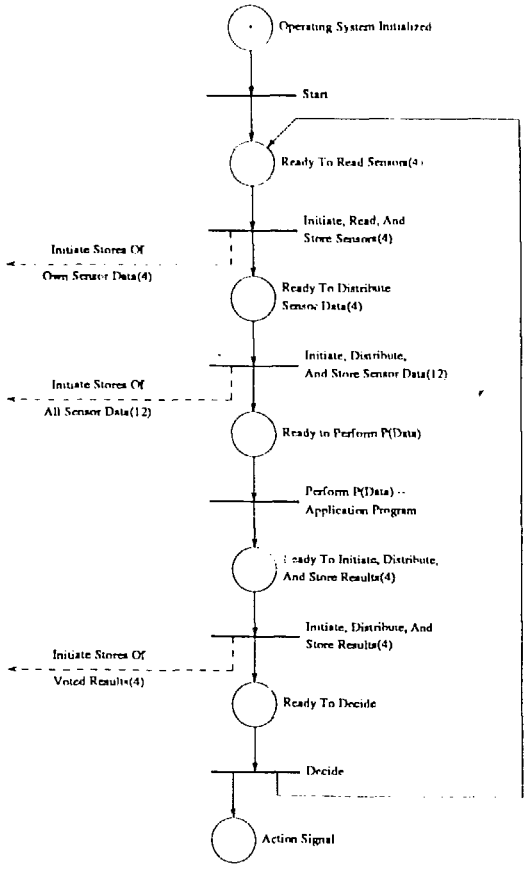

Figure 2.1. A Generic Application Program

The interface at the total system fault-tolerance level (level 1) includes the primary FTP commands that exchange data among the nodes, commands that synchronize the clocks in each node with one another, test and evaluate the status flags for the hardware, and so on. At this phase of the project, we have concentrated on the commands that exchange data and, in particular, how the data transmitted depends on the fault-containment regions. This aspect of the fault-tolerant interface was selected because it is used throughout the operating system (for the fast FDIR and for several of the background tasks), and is related to the interface assumed by the clock synchronizing software. Further, data transmission is an integral part of the fault-tolerance of the application software for reactor flow control.

The interface at the functional level (level 2) includes the data transfer commands between the nodes, the sensor initiate, read, and store commands, and the basic operations of the processor such as the arithmetic operations. Most of the basic operations (except the floating point operations) are common to all 68000-based processors and will be assumed to be working correctly. Other operations such as those associated with sensor data and the arithmetic operations are unique to this application and will be the object of later phases of the FAFTRCS project.

The formal analysis of the software at the second level is based on the inductive-assertions method of Floyd (6). Using the Argonne tools TAMPR (1) and IIP (5), we plan to mechanically transform software, written in the C programming language, using TAMPR to yield those conditions necessary for the software to meet its purported claims. The representation for these conditions, called verification conditions, is the same as that for the hardware, namely Petri nets. An analogous verification condition generator for Fortran implemented using TAMPR has been in operation for the past seven years. The implementation for the C programming language is currently underway.

At this point, the analysis of the interface in terms of these two levels is incomplete. This report discusses progress for the first level as well as the development of an abstraction methodology for specifying the functionality of the second level.

Fundamentals of Formal Ananlysis Based on Automated Reasoning

The automated reasoning system applies a set of axioms to a representation of the system being modeled, and by using a collection of built-in inference mechanisms, attempts to deduce properties of the system. This process implies several considerations. There must be a language in which to represent the axioms and other manipulation rules. The representation used to model a system, in our case Petri nets, must be translated into the language of the reasoning system. And finally, there must be a strategy employed by which proofs of properties are produced.

Several candidate representations have been analyzed. At this time, the decision has been made to use Petri nets as a possible representation (7). It appears that these nets meet the goals of a viable representation. Petri nets are uninterpreted graphs. Our task has been to define efficient operational semantics for them, that is, annotate the graph to give it some functional capability of interest. We have learned that there are many ways to define such semantics, each of which can have drastically different effects regarding proof efficiency, understandability, etc. Clearly, this is an area of continuing research.

Petri nets are composed of circles, called Places, bars, called Transitions, and directed arcs which connect Places to Transitions and Transitions to

Places. In classic Petri net theory, Places may contain an object referred to as a Token. If all Places which are inputs to a Transition contain a Token, then the transition is activated and may fire, which results in all input Places having their Token removed and in all output Places receiving a Token. This simple classic behavior has been extended and modified by various researchers. However, the important point to note is that in terms of computational power, classical Petri nets can be made equivalent to Turing Machines (3).

Basically, we have developed a discipline for the annotation of Petri nets and extended the classic model in order to define the functional capability that is required for our representation of software and hardware (4). For hardware, Places can be used to denote any medium that stores or transports data, for example, a flip-flop, register, bus, etc. Transitions are allowed to represent arbitrary functional modules, such as an adder, an arithmetic logic unit (ALU), a voter circuit, a single logic gate, etc. For software, Places typically represent the status of the actual program counters, and a Token, if present, is the state of the processor at this point. Transitions represent one or more program statements.

For both hardware and software, these restrictions of the use of Places and Transitions are augmented by four extensions to the theory. First, Tokens are allowed to be symbolic expressions which can contain information representing properties of the system. Second, a Place may be specially designated as a source of a token which is read but never removed when a Transition to which the Place is connected fires. Third, Places may be the source of control signals for a Transition. Fourth, Places and Transitions can have a type property associated with them. (The type property for Places and Transitions is discussed further in the context of the proofs of fault-tolerance in (2)). Instead of the simple firing procedure for a Petri net previously described, now the firing will be the symbolic function associated with a transition operating on the symbolic expressions which are the tokens. In addition, the firing will depend on the nature of the control signals which are part of its inputs; and instead of all input Places having their Tokens removed, certain input Places retain their Tokens.

The language of ITP is an extended subset of first-order logic called clausal form. This language is used to represent axioms and the three constructs of Petri nets, namely Places, Transitions, and Connections. The representation is based on templates which are generic in nature, allowing us to provide Transitions with a specific functionality of interest.

The generation of a proof requires the incorporation of strategies. While this report cannot go into the details of such, suffice it to say that the literature is filled with detail concerning these strategies (1). In particular, the researchers at Argonne have developed a considerable body of rules that provide the user of the ITP system with guidance as to the efficient use of the system and with several strategies for generating proofs. However, it must be stated that like first-order predicate logic, clausal form is not decidable. This implies that we cannot always determine whether or not a given property or behavior results from the abstract model of the hardware, or software, system. If the intended result is a logical consequence of the abstract description and the set of axioms and rules that are defined, then this fact will always be established by a formal proof, given enough time. If, on the other hand, the result does not logically follow from the model, a proof cannot always be obtained that formulates this fact: The verification procedure might terminate proving the nonrealization of the property, it might terminate with no result proven, or it might not terminate at all. The inability of not knowing if the verification procedure will terminate reduces to the halting problem of computability theory (3). In cases for which no result is produced, the reasoning system may be able to construct a counter example in order to show that the original conjecture was false. Hence, the use of the reasoning system requires the user to interpret results and to interact with the system. The fact that an inconclusive result can occur is a further reason for our emphasis on a representation that is understandable by the user and that can be easily transformed.

A Generic Application Program

As described earlier, our approach to the modeling and analysis of total system is hierarchical. Consequently, we have formulated a generic application program which has three purposes: to provide a template for writing application programs using three redundant sensors attached to three channels of a quadded FIP; to provide a known and provably correct interface between the hardware and the application program; and to provide a specification for the computational part of the application program that, if followed, assures that the resulting control system is fault tolerant with respect to at most one simultaneous failure of the computing hardware and sensors.

The template for the generic application program is given in Figure 2.1. It specifies that the application program running in each processor cyclically (and simultaneously in each processor) reads its own sensor, that it distributes the sensor value (Sa for Proc-A, Sb for Proc-B, and Sc for Proc-C) to the other processors, that each processor performs a computation, denoted as P(Sa,Sb,Sc), on the three sensor values, and that each processor distributes its result to the other processors in a voted exchange. (Processor d executes the same program but when it reads its own sensor, the sensor-read operation behaves like a null operation and no distribution of that sensor value is performed by processor d, nor is received by the other processors.)

The interface to the hardware occurs in three places. First, each processor's sensor is read. The assumption for this operation is that the processors simultaneously receive a value from three

independent sensors measuring the same physical quantity; however, the values read need not be identical and generally are not the same. Second, the value of each of the three sensors is distributed to the other sensors in a manner that guarantees that the majority of the processors have the same value for that sensor even in the presence of a hardware fault. A formal proof of this assumption using the same automated reasoning tools and representation described here is presented in (2). Third, the result P(Sa,Sb,Sc), computed by the identical application program in each processor, is distributed on a voted exchange to each of the other processors. (The computed result for the applications considered here is a signal which indicates whether the sensor values are nominal or not). If two of the three active (unmasked) processors distribute the same value, then the value returned to each processor is the same and is the majority value, even in the presence of a hardware failure. This result has also been formally proven in (2).

The specification for the computational part of this generic program is that the result P(Sa,Sb,Sc) depends only on the proper operation of the processor in which the computation is performed, assuming that Sa, Sb, and Sc are dependent upon Proc-A, Proc-B, and Proc-C, respectively. This result is proven again with the automated reasoning tools and representation described earlier; the proof is outlined in the next section.

Given the above properties of the hardware/software interface and of the application program, we can readily prove that the majority of the resulting trip signals generated at the Place "Action Signal" of Figure 2.1 is correct, even in the presense of one hardware fault. The proof is outlined as follows: Consider one processor, say Proc-A. (The proof is the same for the other processors.) The acquisition of the sensor values Sa, Sb, and Sc depend upon the proper operation of the Proc-A, Proc-B, and Proc-C, respectively. (It is assumed that the sensors are part of their respective processors.) The exchange of the sensor values makes the modified sensor values Sa', Sb', and Sc' in Proc-A depend on Proc-A, Proc-A and Proc-B, and Proc-A and Proc-C, respectively. This dependency situation now satisfies the conditions for the operation of the application program described in Section 3. There, we have shown that the result P(Sa',Sb',Sc') is dependent on Proc-A. Similarly, the result in the application program in Proc-B and Proc-C is dependent on Proc-B and Proc-C, respectively. Hence, since at most one fault is assumed, at least two of the three processors have the same result. Thus, the conditions for the voted exchange are satisfied, and hence the voted result is the same and correct on each processor that is fault-free. Hence, two of the three action signals are the same and are correct.

The Specific Sensor Validation and Trip Generation Program

A specific application program is illustrated in Figure 2.2. To demonstrate that the entire system is fault tolerant to single hardware and sensor failures, we must show that the result of this program depends only on the processor in which the program is executing.

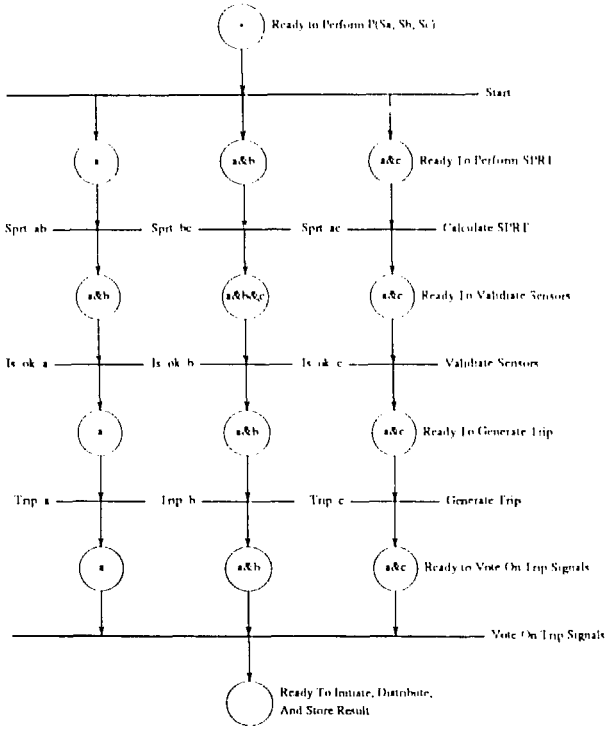

Figure 2.2. A Specific Application Program P(Data)

Before outlining the proof, we need to describe the operation of the program in more detail. On input, the program is given three data values, one for each of three independent sensors reading the same physical quantity. The data values are compared in pairs by a sequential probability ratio test (SPRT). The purpose of this test is to determine if the two values belong to distributions having the same mean and standard deviation. The SPRT is modeled by the three transition labeled Sprt-ab, Sprt-bc, and Sprt-ac in Figure 2.2. Next, the results of the SPRT computation are compared in pairs, yielding three results, indicating the value for sensor a, b, and c are valid; that is, the results of comparing the values for sensors a and b and for sensors b and c are compared -- if either indicate the sensors values are from the same distribution, then sensor a is assumed valid, and similarly for the other pairs of SPRT results. This computation is modeled by the transitions labeled Is-ok-a, Is-ok-b, and Is-ok-c in Figure 2.2. Next, using the validated sensor signals, a trip determination is made by comparing each signal with predetermined ranges of permissible values for the

sensors; this is modeled by the transitions labeled Trip-a, Trip-b, and Trip-c in Figure 2.2. Finally, a vote of three signals is taken on a best two-out-of-three vote. This is modeled by the single transition labeled Vote in Figure 2.2. The result of this vote is the result of the application program, denoted as P(Sa,Sb,Sc).

To prove fault-tolerance, the analysis of the generic application program requires that the result depend upon the correct operation of the hardware in the containment region for the processor which is executing.

The proof of this result has been obtained by the automated reasoning software ITP using a representation for Petri nets of Figure 2.2. The same representation techniques and strategies used to obtain proofs of fault-tolerance for the hardware were used to obtain this proof as well. In particular, the proofs use a general technique of maintaining dependency lists for each operation and datum which avoids the need for lengthy case analysis of all possible single fault scenarios.

The proof of fault-tolerance was based upon explicit assumptions about the signal values, and the computations provided by the SPRT and the trip determination. These assumptions form the basis for a test specification which complements the partition described above. These assumptions are to be validated by either more detailed analysis (such as formal verification techniques described in Section 1.6) or by testing. In this sense, we nvision complementary methods as providing greater completeness than the separate parts as they are connected by a formally defined interface. This completeness has been suggested in reference (Goodenough).

The essence of the proof can be best illustrated by considering the execution of the program of Figure 2.2 on one processor, say c. (The proofs for the program executing in the other processors are the same.) The input signals Sa, Sb, and Sc are dependent upon the correct operation of Proc-A, Proc-B, and Proc-C, respectively. It is assumed that the SPRT yields a result which has a dependency list that is the union of the dependency lists for its inputs. Thus, the dependency lists for the SPRT results are Proc-A, Proc-B, and Proc-C, Proc-A and Proc-C, and Proc-B, and Proc-C for the results denoted Sprt-ab, Sprt-ac, and Sprt-bc. The next operation is a comparison of the SPRT results which in effect indicate which sensor signals are valid. The dependency lists for the results Is-ok-a, Is-ok-b, and Is-ok-c are respectively Proc-A and Proc-C, Proc-B and Proc-C, and processor Proc-C. The operation of the trip determination depends only on the validated signal and the current processor Proc-C, and so the dependency lists for the individual trip signals are the same as for the validation results. Finally, the vote, being a best two-out-of-three vote yields a result which is dependent only on the current processor, namely Proc-C. This property of the best two-out-of-three vote is a consequence of the assumption that there is at most one fault at any given time.

In a similar manner, the results of the programs executing in processors a and b are dependent on Proc-A and Proc-B respectively. This completes the proof.

Conclusions

Using general purpose tools, we have demonstrated that formal analysis of complicated systems can be performed. This paper has discussed the proofs of fault-tolerance of software to failures in hardware, assuming the Draper Fault-Tolerant Processor is used. A previous paper (2) has demonstrated that the same techniques for modeling and analysis can be used for hardware, illustrating that a unified approach for both hardware and software can be used. We have also shown that the software is consistent with the design paradigm for fault-tolerance.

However, a word of caution is appropriate. All of our proofs assume certain properties of the hardware and software environment that have not yet to be proven in the same formal manner. Some of these assumptions will probably never be proven in this manner as the tools are not adequate for such proofs (for example, proofs that the Motorola 68000 performs as specified or that the compiler compiles code correctly). However, the method of proof described herein clearly identifies the assumptions made to obtain the proofs. These assumptions must be examined carefully and validated (accepted or rejected) by whatever criterion is practical.

In summary, we have demonstrated an analysis technique which: Facilitates formalization of a hierarchical specification,

Generates a formal continuity between analysis and test vectors,

Provides a formal communication between designers from varied disciplines.

References

(1) Boyle, J. M., "Program Adaption and Program Transformation," in Practice in Software Adaption and Maintenance, ed, R. Ehert, J. Lueger, and L. Goecke, North Holland Publishing Co., 1980.

(2) Chisholm, G. H., Kljaich, J., Smith, B. T., and Wojcik, A. S., An Approach To the Verification Of A Fault-Tolerant, Computer-Based, Reactor Safety System -- A Case Study Utilzing Automated Reasoning, Research Project 2686-1 Final Report, April 1986.

(3) Hennie, L., Introduction to Computability: Addison-Wesley Publishing Co., Inc., Reading, MA, 1977.

(4) Kljaich, J., "Formal Verification of Digital Systems," Ph.D. Dissertation, Department of Computer Science, Illinois Institute of Technology, Chicago, December 1985.

(5) Lusk, E. L. and R. A. Overbeck, "The Automated Reasoning System ITP," Argonne National Laboratory, Mathematics and Computer Science Division, Report ANL-84-27, April 1984.

(6) Manna, Zohar, Mathematical Theory of Computation, McGraw-Hill, New York, 1974.

(7) Peterson, J. L., Petri Net Theory and the Modeling of Systems: Prentice Hall, 1981.

(7) Rennels, David A., "Fault-Tolerant Computing--Concepts and Examples," IEEE Transactions on Computing, Vol. C-33, No. 12, December 1984.

(8) Smith, T. B., "Data Manipulative Primitives of Synchronous Fault-Tolerant Computer Systems," Proceedings of the 11th Annual International Symposium on Fault-Tolerant Computing, Portland, ME, June 1981.

(9) Smith, T. B., "Synchronous Fault-Tolerant Flight Control Systems," AIAA Computers in Aerospace III, San Diego, July 1981.

(10) Smith, T. B., "Fault Tolerant Processor Concepts and Operation," Proceedings of the 14th Annual International Symposium on Fault-Tolerant Computing, Kissimmee, FL, June 1984.

(11) Wos, L., R. Overbeek, E. Lusk, and J. Boyle, Automated Reasoning: Introduction and Applications: Prentice Hall, 1984.

DISCLAIMER