IN-62

26655

P_129

NASA Contractor Report

Formal Specification and Verification of a Fault-Masking and Transient-Recovery Model for Digital Flight-Control Systems

John Rushby

CONTRACT NAS1-18969
JULY 1991

NASA Contractor Report 4384

Formal Specification and Verification of a Fault-Masking and Transient-Recovery Model for Digital Flight-Control Systems

John Rushby
SRI International
Menlo Park, California

Prepared for
Langley Research Center
under Contract NAS1-18969

National Aeronautics and
Space Administration
Office of Management
Scientific and Technical
Information Program
1991

Contents

Chapter 1

Introduction

This report is concerned with the development and the formal specification and verification of a fault-masking and transient-recovery model appropriate to the replicated computers in digital flight-control systems (DFCS). The culmination of the verification is a mechanically checked theorem which establishes, subject to certain carefully stated assumptions, that faults among the component computers of the DFCS will be masked—so that the commands sent to the actuators will be the same as those that would be sent by a single computer that suffers no failures.

In order to make this report accessible to those unfamiliar with fault-tolerant process-control systems, we begin this chapter with a brief exposition of DFCS, and then present the rationale for the particular model that is the focus of our formal investigation. (See [1] for a full treatment of digital avionics systems, [2] for a treatment of general validation issues, and [3] for a description of current practice in the verification and validation of software for DFCS.)

The second chapter presents the model formally, in the manner of a conventional mathematical development. The proof of the fault-masking and transient-recovery theorem is presented in the same way.

The third chapter outlines the fully formal specification of the model, and its mechanically checked verification. These were undertaken using the EHDM formal specification and verification system [4–8]; the EHDM specification text and related material are given in the Appendices.

The fourth chapter discusses the relationship between the model employed here and the similar one developed by Di Vito, Butler, and Caldwell of NASA Langley Research Center [9, 10]. The fifth and final chapter presents our conclusions and recommendations for further work.

1.1 Digital Flight-Control Systems

Increasingly, modern aircraft rely on Digital Flight-Control Systems—computer systems that interpret the pilot's control inputs and send appropriate commands to the control surfaces and engines.[1] Depending on the aircraft design, DFCS may manage all, or merely some, of the control surfaces and may or may not have back-up systems comprising either analog computers or conventional mechanical and hydraulic systems. The advantages claimed for DFCS include the following:

Safety: DFCS can prevent the pilot stalling the plane, or otherwise taking it beyond its control envelope. For example, the F16 provides yaw-rate limitation to prevent the aircraft entering a certain flat spin mode that has "unacceptable recovery," and rudder fade-out to ensure that "pilots could not get in trouble because of flying habits developed in other aircraft" [11]. Similarly, the Airbus A320 DFCS provides "stall/windshear protection and protection also against overspeed and overstress ... the A320's system automatically prevents the aircraft leaving its safe-flight envelope at any point, whether pilot error or incompetence, engine malfunction, or the elements have brought it to that point" [12] (but see also [13]). Other contributions to safety may include reduction in pilot workload through increased automation and improved handling.

Economy and performance: Elimination of heavy hydraulic and mechanical control linkages reduces aircraft weight and thereby improves fuel-efficiency and load-carrying capacity [14]. Optimum control of engine thrust and angle of attack can also reduce fuel consumption significantly.

Efficiency and performance can sometimes be gained at the expense of handling qualities. DFCS can restore neutral handling characteristics to such aircraft. Maneuverability in unusual flight regimes (e.g., post-stall) may require complex transformations between command inputs and actuator outputs that can only be achieved by computer control. For example, roll commands in the X31 at high angles of attack are interpreted relative to the velocity vector, not the longitudinal axis of the aircraft. Thus at 90° angle of attack, a pure roll command translates to a pure yaw in the body axis [15]. In the limit, high maneuverability, stealth, or other requirements for military aircraft may best be achieved with an unstable airplane—which will require computer control in order to fly at all.

[1] The popular term fly-by-wire (FBW) covers both DFCS and similar, earlier, systems that employ analog computers. Fly-by-light is simply FBW in which fiber-optic cables replace the copper wires used to route signals around the aircraft.

Damage control: The loss of a control surface or engine sometimes results in a crash, not because the airplane is absolutely uncontrollable, but because its pilot is unable to learn how to control it in the time available. For example, it is very hard to control a twin-engined light plane if one of the engines fails, and private pilots often crash in this circumstance. A DFCS can partially compensate for the massive change in flying characteristics caused by failure or damage and thereby assist the pilot to make a safe landing. Simulations have been performed to investigate the efficacy of such systems for military aircraft suffering battle damage [16].

The perceived advantages of DFCS are such that they are employed in almost all modern high-performance western military aircraft. Modern western passenger aircraft generally have full-authority digital engine controls (FADEC); digital autopilot, autolander, and flight management system; and digital control of secondary surfaces and functions, such as air brakes, spoilers, yaw damping, and gust alleviation. However, the Airbus A320 is the only passenger aircraft in service with a full DFCS—that is, one controlling primary control surfaces in the pitch and roll axes.[2] Forthcoming passenger aircraft such as the Boeing 777 will also employ comprehensive DFCS.

The greater the benefit provided by DFCS, the less plausible it becomes to provide adequate back-up systems employing different technologies. For example, the DFCS of an experimental version of the F16 fighter (the "Advanced Fighter Technology Integration" or AFTI-F16) provides control in flight regimes beyond the capability of the simpler analog back-up system. Extending the capability of the back-up system to the full flight envelope of the DFCS would add considerably to its complexity—and it is the very simplicity of that analog system that is its chief source of credibility as a back-up system [17]. Similarly, direct manual control of flight surfaces is unlikely to be available if elimination of heavy mechanical and hydraulic systems was a primary reason for installing DFCS in the first place. Thus, the Airbus A320 has mechanical links to only the rudder and the elevator trimtab [14, 18] and is given no certification credit for these back-up systems by the FAA.

1.2 Fault Tolerance for DFCS

It is clear that extreme reliability must be required of DFCS. A much-quoted figure is a requirement for passenger aircraft that the probability of catastrophic failure during a 10 hour flight should be less than 10^{-9} per hour [19]. Such reliabilities are beyond those that can be guaranteed for individual digital devices. Not only

[2] The Concorde, which received FAA certification in 1969, has analog FBW with mechanical backup in all three primary axes.

must occasional latent manufacturing defects and the effects of aging be considered, but also environmental hazards such as power-supply surges, lightning strikes, and cosmic rays (which can cause single-event-upsets, or SEUs). These factors conspire to yield an overall reliability well below that required. It follows that some form of fault tolerance based on replication and redundancy is needed in order to achieve an underlying "hardware platform" of the required reliability. There are many configurations for redundant and replicated computer systems, and careful reliability analysis is required to evaluate the reliability provided by a given configuration and level of redundancy [20]. Such analyses show that suitably constructed N-modularly redundant systems (which we will call N-plexes for brevity) can achieve the desired reliability.

Within an N-plex, all calculations are performed by N identical computer systems and the results are submitted to some form of averaging or voting. Great care must be taken to eliminate single-point failures, so the separate computer systems (or "channels," as they are often called in fault-tolerant systems) will generally use different power supplies and be otherwise electrically and physically isolated as far as possible. Notice, however, that there is no protection against design faults: any such faults in either the hardware or the software will be common to all members of the N-plex and all will fail together. In this report, we do not address the issue of design faults in the hardware, nor in the application software that it runs. We are, however, very much concerned with the possibility of design faults in the redundancy-management software that harnesses the failure-prone individual components together as a fault-tolerant N-plex. There is evidence (see page 8) that redundancy management is sufficiently complex and difficult that it can become the *primary source of unreliability* in a DFCS.

The function performed by a DFCS is basically one of process control, as portrayed in Figure 1.1. The goal is to control the airplane in flight under command of the pilot. Information about the state of the airplane, which is subject to external disturbances, is obtained through sensors, and control is exercised by sending commands to actuators. The basic structure of most process-control software is very similar: the software performs a repetitive cycle of sampling sensors and control inputs, using control laws to calculate the required actuator response and then sending appropriate commands to the actuators. The complete cycle is generally broken into individual "frames," each attending to a particular dimension of control: for example, one frame may deal with pitch-control—sampling the appropriate sensors, computing the necessary corrections, and sending commands to the elevators; another frame may deal with roll, still another with navigation, and so on. Some variables may need more rapid control than others, so that a complete cycle might contain four pitch-control frames, two roll frames, and only a single navigation frame. This general pattern of activity is described as a multi-rate periodic schedule.

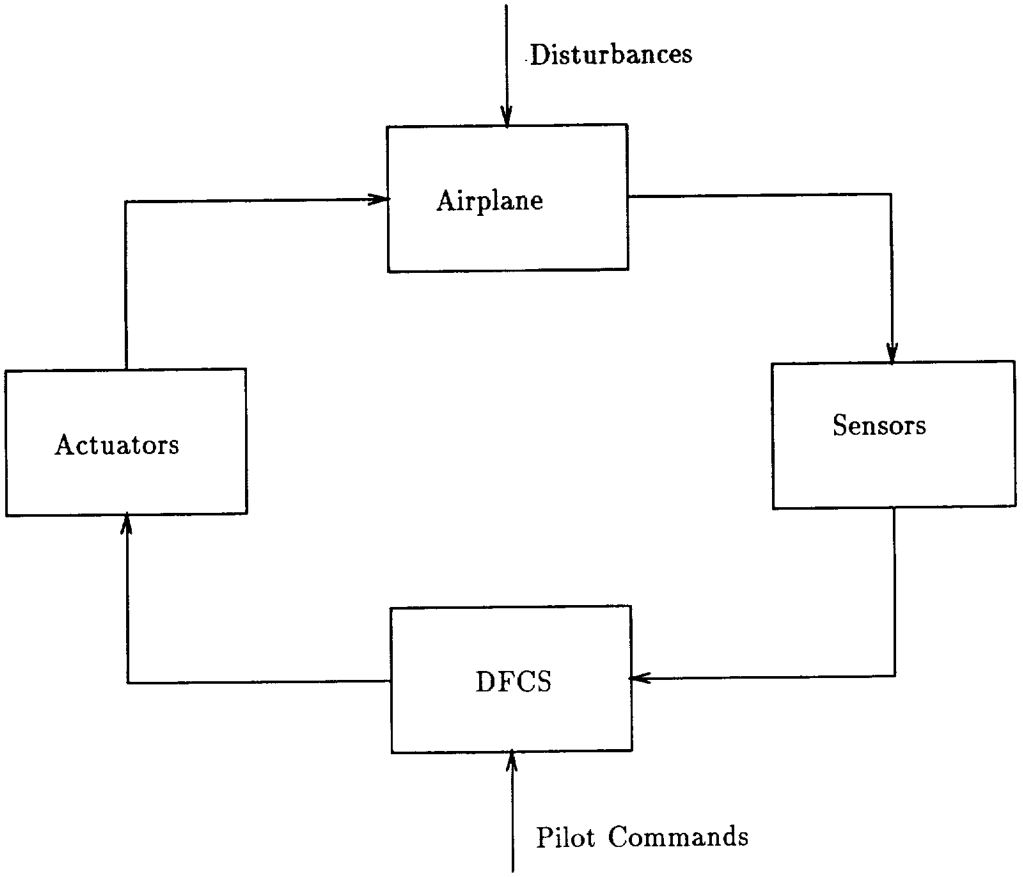


Figure 1.1: The DFCS Process-Control Loop

Each frame will perform several computational activities: sampling sensors, evaluating control laws, generating control outputs, performing self-tests, and so on. "Tasks" are the primitive computational elements within this structure: they are the individual units of activity that may be scheduled and executed. The scheduling slots within a frame and to which individual tasks may be allocated are called "subframes." Thus, for example, the subframes within a pitch-control frame may be allocated to several sensor-sampling tasks, an averaging task to integrate the readings of redundant sensors, a control law task, and an actuator-output task.

Many refinements are possible within this basic paradigm. For example, there may be a fixed, static, schedule of frames, so that all cycles are identical; alternatively, frames may be scheduled dynamically, depending on external circumstances.

Similarly, frames may all execute for a common fixed duration, or may have different durations; they may always execute to completion, or may be subject to preemption, and so on. Whether task scheduling for critical real-time systems should be static or dynamic is a controversial issue. Proponents of static schedules point to *Richards' anomalies* [21,22], in which the early completion of one task can cause another to be late, and other difficulties in dynamic scheduling as indications that the predictability required for hard real-time systems is best achieved by static scheduling.

The major challenge in the design of a fault-tolerant N-plex for DFCS is one of redundancy management. Instead of a single computer executing the DFCS software, there will be several, which must coordinate and vote (or average) actuator commands,[3] and tolerate faults among their own members. In addition to the replicated computers, sensors and actuators will be replicated also. The management of all this redundancy and replication adds considerable complexity to both the operating system (generally called an "executive" in process-control systems) and the application tasks. Complexity is a source of design faults, and there is a distinct possibility that such a large quantity of additional code may lessen, rather than enhance, overall reliability. The goal of the research program, of which this work is a component, is to develop principled, structured, and formally specified and verified approaches to the design and implementation of redundancy management in DFCS [9].

A plausibly simple approach to redundancy management in an N-plex is the "asynchronous" design, in which the channels run fairly independently of each other: each computer samples sensors independently, evaluates the control laws independently, and sends its actuator commands to an averaging or selection component that chooses the value to send to the actuator concerned. The triplex-redundant DFCS of the experimental AFTI-F16 was built this way, and its flight tests reveal some of the shortcomings of the approach [17,23].

First, because the unsynchronized individual computers may sample sensors at slightly different times, they can obtain readings that differ quite appreciably from one another. The gain in the control laws can amplify these input differences to provide even larger differences in the results submitted to the output selection algorithm. During ground qualification of the AFTI-F16, it was found that these differences sometimes resulted in a channel being declared failed when no real failure had occurred [24, p. 478].[4] Accordingly, rather a wide spread of values must be accepted by the threshold algorithms that determine whether sensor inputs and actuator outputs are to be considered "good." For example, the output thresholds

[3] Voting or averaging is often performed directly by the actuators, through some form of "force-summing." For example, different channels may energize separate coils of a single solenoid, or multiple hydraulic pistons may be linked to a single shaft [11, Figure 3.2-2].

[4] Also, in the flight tests of the X31 the control system "went into a reversionary mode four times in the first nine flights, usually due to disagreement between the two air data sources" [15].

of the AFTI-F16 were set at 15% plus the rate of change of the variable concerned; also the gains in the control laws were reduced. This increases the latency for detection of faulty sensors and channels, and also allows a failing sensor to drag the value of any averaging functions quite a long way before it is excluded by the input selection threshold; at that point, the average will change with a thump [23, Figure 20] that could have adverse effects on the handling of the aircraft.

The danger of wide sensor selection thresholds is dramatically illustrated by a problem discovered in the X29A. This aircraft has three sources of air data: a nose probe and two side probes. The selection algorithm used the data from the nose probe provided it was within some threshold of the data from both side probes. The threshold was large to accommodate position errors in certain flight modes. It was subsequently discovered that if the nose probe failed to zero at low speed, it would still be within the threshold of correct readings, causing the aircraft to become unstable and "depart." This error was found in simulation, but 162 flights had been at risk before it was detected [25].

An even more serious shortcoming of asynchronous systems arises when the control laws contain decision points. Here, sensor noise and sampling skew may cause independent channels to take different paths at the decision points and to produce widely divergent outputs. This occurred on Flight 44 of the AFTI-F16 flight tests [23, p. 44]. Each channel declared the others failed; the analog backup was not selected because the simultaneous failure of two channels had not been anticipated and the aircraft was flown home on a single digital channel. Notice that all protective redundancy had been lost, and the aircraft was flown home in a mode for which it had not been designed—*yet no hardware failure had occurred.*

Another illustration is provided by a 3-second "departure" on Flight 36 of the AFTI-F16 flight tests, during which sideslip exceeded 20°, normal acceleration exceeded first −4g, then +7g, angle of attack went to −10°, then +20°, the aircraft rolled 360°, the vertical tail exceeded design load, all control surfaces were operating at rate limits, and failure indications were received from the hydraulics and canard actuators. The problem was traced to an error in the control laws, but subsequent analysis showed that the side air data probe was blanked by the canard at the high angle of attack and sideslip achieved during the excursion; the wide input threshold passed the incorrect value through and different channels took different paths through the control laws. Analysis showed this would have caused complete failure of the DFCS and reversion to analog backup for several areas of the flight envelope [23, pp. 41–42].

Several other difficulties and failure indications on the AFTI-F16 were traced to the same source: asynchronous operation allowing different channels to take different paths at certain selection points. The repair was to introduce voting at some of these "software switches." In one particular case, repeated channel failure indications in flight were traced to a roll-axis "software switch." It was decided to vote the switch (which, of course, required *ad hoc* synchronization) and extensive simulation and testing were performed on the changes necessary to achieve this. On the next flight, the problem was found still to be there. Analysis showed that although the switch value was voted, it was the unvoted value that was used [23, p. 38].

The AFTI-F16 flight tests revealed numerous other problems of a similar nature. Summarizing, Mackall [23, pp. 40–41] writes:

> "The criticality and number of anomalies discovered in flight and ground tests owing to design oversights are more significant than those anomalies caused by actual hardware failures or software errors.
>
> "...qualification of such a complex system as this, to some given level of reliability, is difficult ...[because] the number of test conditions becomes so large that conventional testing methods would require a decade for completion. The fault-tolerant design can also affect overall system reliability by being made too complex and by adding characteristics which are random in nature, creating an untestable design.
>
> "As the operational requirements of avionics systems increase, complexity increases. Reducing complexity appears to be more of an art than a science and requires an experience base not yet available. If the complexity is required, a method to make system designs more understandable, more visible, is needed.
>
> "The asynchronous design of the [AFTI-F16] DFCS introduced a random, unpredictable characteristic into the system. The system became untestable in that testing for each of the possible time relationships between the computers was impossible. This random time relationship was a major contributor to the flight test anomalies. Adversely affecting testability and having only postulated benefits,[5] asynchronous operation of the DFCS demonstrated the need to avoid random, unpredictable, and uncompensated design characteristics."

It is difficulties such as these that have caused those performing research in fault-tolerant systems for DFCS to prefer synchronized channels and exact-match voting [26–28]. Of course, the synchronization must itself be fault-tolerant and no such algorithms were known until about 1982.[6] A number of provably correct Byzantine fault-tolerant clock synchronization algorithms are now available [32–37],

[5] The decision to use an asynchronous design for the AFTI-F16 DFCS was because "the contractor [Bendix under subcontract to General Dynamics] believed synchronization would introduce a single-point failure caused by electromagnetic interference (EMI) and lightning effects" [23, p. 7]—which may well have been correct given the technology of the early 1980s.

[6] Prior to the investigations of the SIFT project [29], the subtlety and delicacy of voting and synchronization protocols were not properly understood and most were seriously flawed: all were vulnerable to Byzantine faults (which constitute a fault class that had not been recognized before), and many were incapable of tolerating less severe faults. For example, the failure of the first attempt

and some have been formally verified [38]. An algorithm due to Infis and Moore [39] is attractively simple, and tolerates a very wide class of faults that is, however, short of the fully Byzantine. Probabilistic algorithms due to Cristian [40] can achieve very close synchronization, but also fall short of Byzantine fault tolerance.

For exact-match voting, each channel must operate on the same data. Thus the computers cannot simply use their own private sensor readings, but must exchange sampled values with each other in a Byzantine fault-tolerant manner. By this means, every (working) computer begins each frame with the same set of sensor readings as the others.[7] Each computer will then run the same sensor selection and averaging algorithms,[8] and the same control laws, and should therefore generate identical actuator commands. Exact-match majority voting of the actuator commands then suffices to mask faults among the redundant channels. Notice that this arrangement allows sensor failures to be distinguished from failures among the redundant computers: sensor failure is detected or masked by the diagnostic, averaging, and selection algorithms run by each computer, whereas failure of a computer is masked (and optionally detected) by the exact-match majority voting of their outputs. In contrast, systems based on unsynchronized, independent channels cannot distinguish accurately between the failure of a sensor and that of a computer, and may mistake the consequences of clock drift for either.

Majority voting of actuator commands is sufficient to tolerate up to $\frac{N-1}{2}$ faults. However, the underlying Byzantine fault-tolerant clock synchronization and interactive consistency algorithms can tolerate only $\frac{N-1}{3}$ faults: thus a 4-plex is required for single-fault tolerance, and a 7-plex for two-fault tolerance. Notice, however, that the 7-plex can withstand two *simultaneous* faults; if the fault arrival rate is such that a faulty channel can be identified and configured out of the system before the next fault arrives, then a 7-plex can withstand 4 faults, and two-fault tolerance can be achieved by a 5-plex. Fault detection and reconfiguration are complex functions, however, and given our desire to reason formally about fault-tolerance properties, we follow [9] and consider only the nonreconfigurable case in this work. (Reconfiguration was considered in the verification of SIFT [53].)

Not all faults are equal: some are "hard" faults that permanently disable the afflicted channel; others are "soft" or "transient" faults from which recovery is pos-

to launch the Space Shuttle was due to a synchronization problem [30], and the heavy radiation environment at Jupiter caused loss of synchronization on the Voyager spacecraft [31].

[7] A given sensor may be sampled independently by several computers; all of these independent samples must be distributed to all other computers in a Byzantine fault-tolerant manner. As with clock synchronization, several Byzantine agreement (or interactive consistency) algorithms are known [41], and some have been formally verified down to the hardware implementation level [42,43].

[8] In addition to detecting faults, the processing of sensor data must deal with noise, bias, drift, hysteresis, and other sensor-specific issues. The problems of sensor averaging, selection, and (especially) fault diagnosis have been considered, more or less independently, by several disciplines—for example, control theory [44–49], artificial intelligence [50,51], and computer science [52].

sible. Examples of transient faults include SEUs (where a single bit of memory is flipped by a cosmic ray), which can be recovered by simply restoring the affected bit to its correct value. Experience indicates that transient faults are orders of magnitude more common than hard faults—for example, Voyager spacecraft suffered 42 SEUs in the intense radiation surrounding Jupiter, but no hard faults [54]. It follows that overall reliability will be much greater—or, equivalently, much less redundancy will be required for a given level of reliability—if some attempt is made to recover channels that suffer transient faults.

There is no firm line between transient and hard faults considered in the abstract; what might be merely a transient fault to one system may be a hard fault to another that lacks the necessary recovery mechanisms. Fault-tolerant system architectures are designed and evaluated against explicitly stated fault models. For transient faults, we employ a fault model in which we distinguish two subclasses of faults.

State data faults are those in which the processor is working correctly (i.e., is synchronized and executing the right task), but its local state data are corrupted. If its state data were replaced with correct values, it would recover. In our formal model, the predicate $OK(i)(c)$ will indicate whether processor i has state data faults that can affect its computation of task c.

Control faults are those in which the processor is not working correctly (i.e., something other than, or additional to, a state data fault has occurred). In our formal model, the predicate $\mathcal{F}(i)(j)$ will indicate whether processor i suffers a control fault during the computation of the j'th task.

In our model, we think of control faults as happening spontaneously, and state data faults as the consequences of control faults. Faults such as SEUs, in which a single bit of state data is spontaneously corrupted, can be considered as instantaneous control faults: we imagine that the processor computes the wrong value but then immediately recovers, leaving a state data fault behind. Note that a state data fault may precipitate a further control fault. For example, a word of memory may become set to zero (a state data fault); then a subsequent divide operation using that word might generate a divide-by-zero trap, which could halt the processor (a control fault).

State data faults can be recovered by periodically replacing the state data maintained by each processor with a majority-voted version. It is not necessary to vote and replace all the state data, since many of them are refreshed by sampling sensors (i.e., some of the state data are "stored" in the airframe itself [18]): only the data that are carried forward from one frame or cycle to the next (e.g., time-integrated data such as velocity and position) need to be voted. Even so, the quantity of state data maintained by a modern DFCS is considerable, and performance would be seriously degraded if all of it were voted at every opportunity. Accordingly, exposure is traded for performance and rather sparse voting patterns are preferred. Clearly the less frequently a particular item of state data is voted, the longer will be the duration of the consequences of a fault that corrupts that item. Overall reliability will be determined by the fault arrival rate, the voting pattern, and the dataflow dependencies among control tasks and state data items.

In this report, our goal is to develop, and formally specify, a model that describes the operation of an N-plex with transient-recovery based on an arbitrary sparse voting pattern. We will formally verify a theorem concerning the conditions under which such a system masks faults successfully. A concrete instance of the theorem (for a specific data dependency graph and voting pattern) might be that the system is "safe" provided that at most two processors suffer control faults in any sequence of five successive frames. Markov or other methods of reliability analysis must be used to determine the overall reliability of the system, given assumptions about the arrival and repair rates of control faults [9].

A fault-tolerant system should take active measures to recover from transient control faults, in addition to the voting strategy for overcoming state data faults. The Mars system [55,27] is a good example of a system that provides such recovery. In our model, however, we do not consider the internal details of mechanisms that achieve recovery from control faults, we model only their external behavior; the purpose of our model is to derive properties of the majority voting strategy for masking faults of all kinds and recovering from state data transient faults.

1.3 Formal Models for DFCS

In this section we sketch the larger context of the work described here, and then give an overview of the model for fault-masking and transient-recovery that we employ.

This work was performed in the context of a research program led by NASA Langley Research Center that aims to develop a fault-tolerant architecture for DFCS using formal methods to provide a rigorous basis for documenting and analyzing design decisions. Ultimately, we hope to provide mechanically-checked formal specifications and verifications for the key components of a "Reliable Computing Platform" for DFCS, going all the way from high-level requirements down to implementation details. Clearly, this is a major undertaking, so initially we are concentrating on some of the better-understood requirements and levels in the hierarchy.

As we described in the previous section, synchronized channels and Byzantine fault-tolerant distribution of sensor values are now fairly well-understood requirements. Accordingly, the first mechanically-checked specifications and verifications undertaken in this program were those performed for Byzantine fault-tolerant clock synchronization algorithms [38,56] and for a Byzantine agreement algorithm [42] and circuit [43]. The work described here is a step towards the next higher layer in the modeling hierarchy: the layer that uses exact-match voting to provide fault-tolerance and transient-recovery.

Accurate modeling of that layer must account for the fact that the separate channels are not perfectly synchronized (the clock-synchronization algorithms keep the separate channels synchronized only within some small skew δ of each other), and that the communication and coordination of voting data takes a certain amount of time. The work presented here ignores those details in order to concentrate on the relationship between voting patterns, fault masking, and transient recovery. Thus, we make the simplifying assumptions that the separate channels are perfectly synchronized, and that the communication and voting of data constitute a single atomic action.[9]

Our current work, following on from that described here, aims to eliminate these simplifying assumptions. In other current work, we are developing and formally verifying a hardware-assisted implementation of one of the clock-synchronization algorithms. Future work may consider the mechanisms by which failed channels can be recovered, or the system reconfigured. The next section gives an informal overview of the model that is the focus of the present analysis.

1.3.1 Overview of the Fault-Masking Model Employed

In companion work at NASA Langley Research Center, Di Vito, Butler and Caldwell [9] have developed a formal model for DFCS and derived its fault-masking and transient-recovery properties. Their model and development is formal and rigorous in the manner of conventional mathematical discourse. The purpose of our investigation is to construct a completely formal, machine-checked specification for a similar model, and to submit the derived properties to mechanical proof-checking. The two investigations are complementary: the first is intended to model the structure of a realistic platform for DFCS, while the second is intended to explore the problems of subjecting formal specifications and verifications in this domain to mechanically checked analysis.

Our model for fault masking and transient-recovery was developed in parallel with that of [9] and differs from it in several details, though not in overall principle. In this section, we briefly sketch the model of Di Vito, Butler, and Caldwell, and explain how and why ours differs. The relationship is described in more detail in Chapter 4.

Di Vito, Butler, and Caldwell model a reliable computing platform for DFCS with the following characteristics:

- The system workload is a multi-rate periodic schedule.

[9] Verification of the Oral Messages Byzantine agreement algorithm [42, 43] makes the same simplifying assumptions.

- The schedule is static (i.e., the sequence of frames is identical from one cycle to another, and the subsequence of tasks within a given frame is the same in every activation of that frame).

- All frames have equal duration; however, different frames may have different numbers of tasks, and different tasks may have different duration. Unused time at the end of a frame is called "slack" time; it can be used to run self-tests. Some slack time at both the beginning and the end of each frame is essential when discrete-update clock synchronization is used, since otherwise tasks could be skipped (if the clock jumps forward) or repeated (if it jumps back) [34].

- The output of a task may be used as input to a later task up to one cycle later. Data that need to be carried further forward must be relayed through intermediate tasks.

- Sensors are sampled and actuators commanded at most once per frame. An underlying Byzantine fault-tolerant distribution of sensor samples is assumed, so that each (working) channel receives identical sensor input.

- The fault model distinguishes processors that are working correctly throughout a frame from those that are not. In our terminology, correctly working processors, or more briefly, *working* processors, are those without control faults. A *fault-status* predicate indicates whether a given processor is working or not in the current frame. Faults can be either permanent (i.e., hard) or transient—the latter is modeled by a processor whose *fault-status* is *not working* in one frame and *working* in a later one. The model does not consider the mechanisms by which such recovery might be achieved.[10]

- Various voting patterns are considered. In *continuous* voting, all state data are voted every frame; in *cyclic* voting, only the outputs of tasks in the current frame are voted in that frame; *minimal* voting uses the dataflow dependencies among tasks to derive conditions that vote the minimum data each frame. A distinguished state data item, the *frame-counter* is always voted at every frame.

- All processors run identical workloads. The benchmark with respect to which fault-masking and transient-recovery results are proved is a single processor running the same workload that suffers no faults.

[10] Among the likely mechanisms are watchdog timers that trap to automatic re-initialization code, and similar reinitialization of the losers in a majority vote. In addition, the schedule table and the object code for the system executive and application tasks may be held in ROM, where all faults may be assumed hard, but also extremely rare.

Our model is very similar in spirit and motivation to that just described; it differs in being considerably more abstract. The reason for this is that we want our results to be as widely applicable as possible. Mechanically checked formal specification and verification are very time consuming to perform and mechanically checked proofs tend to be rather fragile. By this we mean that redoing a mechanically checked proof to accommodate changes to the statement of a theorem, or modifications to supporting lemmas, may require a quantity of effort and insight comparable to that required to construct the proof in the first place. Thus it is often not cost-effective to prove properties about a variant model by adjusting proofs from the original model. A generally more productive approach is to employ abstraction and hierarchy: one attempts to extract the essence of the problem and to prove the most general results possible for an abstract formulation of the problem. More concrete models can then be constructed as instantiations or elaborations of the abstract model, and properties concerning the elaborations can be proved using the abstract theorem as a lemma.

In the present case, we obviously wish to derive results that are sufficiently general that they can apply to all three of the voting schedules considered by Di Vito, Butler, and Caldwell. We would also like them to be applicable to systems that make rather different basic assumptions—for example, systems in which sensors are sampled and actuators commanded more then once per frame, or in which not all cycles have identical frame schedules (so that dynamic scheduling can be accommodated). We wish to state and prove general results along the lines of "provided the voting strategy satisfies certain properties, and providing certain fault assumptions are met, then an N-plex correctly masks faults and recovers from transients."

A little thought reveals that the essence of this problem concerns the interaction between voting strategies, task schedules, and data dependencies. To see this, consider a particular actuator command. We want the majority value for this command to equal the "correct" value (i.e., that which would be produced by a single fault-free processor). Clearly, this will be so if a majority of processors are working correctly at the time they execute the task concerned *and* if they receive the correct input values. Input values either come from sensors (and our requirement here is that all working processors receive the same values), or they are the outputs of previous tasks, which may or may not have been voted. In the case of voted outputs, we recurse on the conditions that establish the correctness of voted outputs; in the case of nonvoted outputs, the requirement is that the majority were working correctly when that task was executed, and that their inputs were, in turn, correct at that point. Obviously a development along these lines must make very careful statements about its assumptions, and there are many tricky details to be taken care of, but it is equally obvious that the notions of cycles and frames are not essential to the argument: it is the order in which tasks are executed, the dataflow dependencies among them, and the placement of majority votes that determine the correctness of the overall scheme.

Thus, frames and cycles are not explicitly represented in our model: we represent the system workload by the dataflow dependency graph among task activations,[11] and a record of the order in which tasks are activated. We allow voting to be specified for the outputs of any task activation, and we model processor failure at the task activation level (i.e., a given processor is either *working* or *not working* during a given task activation). It should be clear that a periodic, frame-based interpretation can be achieved by simply imposing additional structure on the task activation dataflow dependency graph and on the task execution schedule. (For example, by requiring them to have a periodic structure, allowing only one voted task per frame, treating failure during any task activation as a failure for the whole frame, and so on.) In this way, results proven for our abstract model provide a basis for deriving results for more concrete models relatively easily.

In addition to cycles and frames, we have abstracted away another aspect of the model of Di Vito, Butler, and Caldwell: the frame-counter. Some may consider our use of abstraction to have been overly aggressive in this regard. Our original motivation was as follows. For a given processor to compute the correct outputs for a certain task activation, it must be working correctly during that task, and it must get the correct inputs. Whether it gets the correct inputs is a function of when data were voted, and of how long the processor has been working correctly. Here, "working correctly" means correctly executing the right programs at the right time, but on possibly corrupted data—i.e., it is the absence of control faults. We do not model the mechanisms by which a processor that has been *not working* (i.e., has suffered a transient control fault) gets back into the *working* state (i.e., recovers from the control fault). Part of this process may involve purging internal corruption (e.g., a stuck-at carry-flag) by means of a system reset, or a power cycle. Another part may involve reloading external state data (such as the identity of the current point in the task schedule—i.e., the frame counter). Surely, reloading this datum is simply part of the internal process of recovery from a control fault, and is therefore part of an activity that we have explicitly chosen not to model.

A counter-argument to this position would observe that the only reliable source for such external data is the majority-voted consensus of the other processors. Thus, this part of the process for recovery from control faults depends on the voting strategy and on the mechanism for recovery from state data faults—the very core of what we have chosen to model. We are partly persuaded by this argument, but note that the data concerned differ from other state data treated within the model in that they are not produced and consumed by application tasks but by the system

[11] A *task* properly refers to a particular program, viewed as a static entity (e.g., as a sequence of bytes, or as a function from inputs to outputs), a task *activation* refers to an instance of that program in execution. There is only one instance of each task, but it gives rise to many activations. Sometimes, when the context makes the intended interpretation clear, we use the shorter term task to mean task activation.

executive itself. On the other hand, we are not attracted to a special-case treatment of the frame counter—if other system state data needed to be recovered in a similar way, another special-case adjustment to the model might be needed.

Our current preference thus remains the exclusion of the frame counter from our basic system model. However, the frame counter (and other state data used by the executive rather than by the application tasks) can be introduced quite simply and naturally when the model is instantiated: simply introduce a voted task (interpreted as the vote of the frame counter and other system state data) at the beginning of each frame[12] and introduce a data dependency of all other tasks within the frame on the output of that particular task. This last is an artifact of the model (in that no real dataflow need occur), but serves to establish the (control) dependency of subsequent tasks upon the correctness of the value for the frame counter obtained by (the task corresponding to) the vote on its value. The task that votes the frame counter is understood to be a standard task performed by all (synchronized) processors at frame-start time, independently of whether they (already) know what frame it is they should be executing.

Although the modeling is indirect, this approach allows the properties of systems with a voted frame counter to be derived correctly, while preserving the abstractness—and hence the wider applicability—of our model. Unlike special-case treatment for the frame counter, our approach easily accommodates more or less frequent voting of this value, and the introduction of additional state data that are required for the correct execution of the executive itself.

In Chapter 2, we present the details of our fault-masking model in the form of a traditional mathematical development.

[12] In [9], all voting occurs at the *end* of each frame; thus, in this case, the identity of the current frame is recovered by the vote at the end of the previous frame. Clearly, our approach can be adjusted to accommodate this alternative arrangement.

Chapter 2

The Fault-Masking Model

Our goal is to prove that, subject to certain conditions, an N-plex provides transient-recovery and fault masking for a certain class of faults. Our first requirement, therefore, is a benchmark model for correct, fault-free behavior, against which the efficacy of transient-recovery and fault masking in the N-plex may be judged. We take as our benchmark a model for the behavior of a fault-free process-control system. Our model for an N-plex will then compose N fault-prone versions of the basic model, together with some voting and recovery mechanisms, and our theorem will establish that the voted results of the N-plex equal those of the fault-free system (under suitable conditions). We begin by describing our model for fault-free process control.

2.1 A Model for Fault-Free Process Control

A process-control system manages some physical device by sending control signals to *actuators*. The values of the control signals are determined by calculations based on the values of *sensors* that monitor the device and on a record (maintained by the process-control system) of the *state* of the system. The process-control system is internally composed of computational *tasks* that are activated periodically in order to sample sensors, perform the necessary calculations, and send values to the actuators. Some tasks may also perform internal housekeeping functions. Because task activations may depend on the results of other task activations, there is a dataflow dependency among task activations that the execution schedule must take into account. The "slots" in the execution schedule are called *cells*[1]; a process-control system requires a specification of which tasks are assigned to which cells, the dataflow relationships among cells, and the order in which cells are to be executed. These ideas are formalized in the following definitions.

[1] In a frame-based system they are often called *subframes*.

We assume

- A set C of *cells*, and
- A relation $G \subseteq C \times (\mathbf{N} \times C)$ (where $\mathbf{N}$ denotes the natural numbers),

and we define

- $M \stackrel{\mathrm{def}}{=} \{1, 2, \ldots, |C|\}$.

Cells correspond to the activations (or executions) of *tasks* (to be formally defined later) or the sampling of sensors; the relation G records the dataflow dependencies among task activations associated with cells: the interpretation of $(i, (n, j)) \in G$ is that the output of the task activation (or sensor sample) associated with cell i supplies the input for the n'th argument of the task activation associated with cell j. A simplified relation

- $\overline{G} \stackrel{\mathrm{def}}{=} \{(i, j) | \exists n : (i, (n, j)) \in G\}$

captures just the basic dataflow dependencies among cells, without concern for which input of cell j it is that receives its data from i. We will ensure by conditions given later that $\overline{G}$ is a directed acyclic graph—so that there are no circularities in the dependencies among cells.

Note that the set C of cells comprises all the task activations performed during a single run of the system (which may extend for the entire lifetime of the system). It is therefore potentially unbounded (though finite) in size. For many (statically scheduled) process-control systems, the set C and its associated data dependency graph $\overline{G}$ will have a repetitive structure induced by the "unrolling" of a periodic, or cyclic, pattern of activity.

Cells with indegree zero in $\overline{G}$ are called *sensor* cells; those with outdegree zero are called *actuator* cells. The set of sensor cells is denoted C_S; that of actuators is denoted C_A. Nonsensor cells (including actuator cells) have a computational task associated with them and are called *active-task* cells. The set $C \setminus C_S$ of active-task cells is denoted C_T.

Each task activation (or sensor sample) generates a value that is either communicated to an actuator or stored so that it will be available as input to later task activations. The system state records these stored output values. Formally, we define

- A set D of *domain* values, and
- A set of *states* $S \subseteq C \to D$.

The data values computed, stored, and manipulated by the system are assumed to be drawn from the uninterpreted domain D. The system state is represented by a function from cells to this domain: if $\sigma \in \mathcal{S}$ is the instantaneous state of the system, and c is a cell, then $\sigma(c)$ denotes the output value stored for that cell. It may seem that a system satisfying this description must have a huge amount of storage in order to record the values of all task activations for all time. This is not so. Anticipating definitions that are given below, we observe that tasks are executed in a sequential order that respects the dependency ordering represented in the graph $\overline{G}$, and run to completion. There is no need to record a value for a cell that has not yet been executed, nor for one whose immediate successors in the relation $\overline{G}$ have already completed. Although this result is intuitively obvious, its formal verification is an interesting exercise (see page 47).

Formalizing the notion of sequential execution, we introduce

- A bijection $sched: M \rightarrow C$, with
- Inverse $when: C \rightarrow M$.

The interpretation here is that the i'th task execution (or sensor sample) is the one associated with cell $sched(i)$; conversely, the activity at cell c is the $when(c)$'th to be executed. We require that the order of execution respect the dataflow dependencies recorded in $\overline{G}$:

$$(i,j) \in \overline{G} \supset when(i) < when(j).$$

Notice that this requires that $\overline{G}$ is acyclic.

Active-task cells have some computational task associated with them, so we require

- A set $T \subseteq \mathcal{S} \rightarrow D$ of *task-functions*, and
- A function $task: C_T \rightarrow T$.

When an active-task cell c executes, the function $task(c)$ is applied to the current state, say σ, yielding the result $task(c)(\sigma)$. This is then stored in the system state as the value of cell c to yield a new state τ. That is,

$$\tau = \sigma \textbf{ with } [c := task(c)(\sigma)]$$

where **with** [...] denotes function modification (as in EHDM).[2] The only components of the system state that may influence the result are those of the immediate

[2] The notation f **with** $[x := a]$, where x is a value in the domain of f and a a value in the range, denotes a function with the same signature as f defined by

$$f \textbf{ with } [x := a](y) = \textbf{if } x = y \textbf{ then } a \textbf{ else } f(x).$$

predecessors of cell c in the dataflow dependency graph $\overline{G}$.[3] Formally, we state this as a requirement that the result be functionally dependent on just those values:

$$(\forall (a, c) \in \overline{G} : \sigma(a) = \tau(a)) \supset task(c)(\sigma) = task(c)(\tau).$$

Sensor cells store their results in the system state just like active-task cells. However, they take no input from the system state; instead, they sample properties of the external environment (including control inputs). These properties vary with time, so it might seem that sensors should be modeled as functions of real-time. In fact, this is unnecessary and inappropriate, since our model is not concerned with real-time properties such as absolute execution rates, but with those of sequencing and voting. We want to prove that if an N-plex gets the *same* sensor samples as an ideal fault-free system, then it will deliver the same actuator commands (despite the occurrence of faults). Thus, we need only model the sensor samples actually obtained, which can be done by modeling sensor samples as functions of position in the execution schedule (i.e., we use the number of cells executed as our notion of "time"). Thus we introduce

- A set $S \subseteq M \rightarrow D$ of *sensor-functions*, and
- A function $sensor \colon C_S \rightarrow S$.

When a sensor cell c executes, the sensor-function $s = sensor(c)$ samples the environment (at time $when(c)$) to yield the value $s(when(c))$. This is then stored in the system-state as the value of cell c.

Formally, the execution of cells is modeled by the function

- $step \colon \mathcal{S} \times C \rightarrow \mathcal{S}$

where

$$step(\sigma, c) \stackrel{\text{def}}{=} \sigma \textbf{ with } [c := \textbf{if } c \in C_S \textbf{ then } sensor(c)(when(c)) \textbf{ else } task(c)(\sigma)]$$

is the new state that results from executing the task of cell c in state σ at time $when(c)$.

We are interested in the state after the system has executed some number m of cells according to its schedule. This is modeled by the function

- $run \colon M \rightarrow \mathcal{S}$

[3] Operationally, the function $task(c)$ is applied to the tuple of values

$$(\sigma(c_1), \sigma(c_2), \ldots, \sigma(c_n))$$

where $(c_i, (i, c)) \in G$ and $n = indegree(c)$.

where

$$\begin{aligned} run(0) &\in \mathcal{S}, \\ run(m+1) &\stackrel{\text{def}}{=} step(run(m), sched(m+1)). \end{aligned}$$

A variant is the function

- $runto: C \rightarrow \mathcal{S}$

where

$$runto(c) \stackrel{\text{def}}{=} run(when(c))$$

is the state of the system when execution of its schedule has reached cell c. Observe that $run(0)$, the initial state, is chosen arbitrarily.

2.2 The N-plex Model

In this section, we admit the possibility that machines may fail and we introduce replication and voting to overcome that fallibility.

We assume a replicated system comprising $r \geq 3$ component systems of the type described in the previous section and we define

- $R \stackrel{\text{def}}{=} \{1, 2, \ldots, r\}$.

In the following, we will often refer to the component systems as "machines."

Component machines may fail and revive independently; at any time a machine is either "failed" or "working." This is specified by a function

- $\mathcal{F}: R \rightarrow (M \rightarrow \{T, F\})$

where $\mathcal{F}(i)(m)$ is T just in case component machine i is failed at time m.[4] Intuitively, a component machine i is failed at time m if it suffers a control fault at any point during execution of the task scheduled at time m. We know nothing at all about the behavior of failed component machines. Working (i.e., non-failed) machines correctly compute the function associated with the task scheduled at time m. However, the result computed may be incorrect if an earlier failure has caused the input data to be bad. A machine that is working correctly, but on bad data, has state data faults that will eventually be overcome through majority voting of state data.

States of the replicated machine are drawn from the set

- $\mathcal{R} \subseteq R \rightarrow \mathcal{S}$.

[4] A function with range $\{T, F\}$ can be interpreted as the characteristic predicate of a set (this is how sets are defined in EHDM). Thus $\mathcal{F}(i)$ can be interpreted as the set of times when the i'th machine is failed during execution of the cell scheduled at that time.

Thus, if $\rho \in \mathcal{R}$ is a replicated state, then $\rho(i)$ is the state of the i'th component machine, and $\rho(i)(c)$ is the value of cell c in that machine.

The components of a replicated machine behave much like a single machine, except that components may fail, and so they periodically *vote* their results. Thus we assume a set

- C_V of voted cells

and require

$$C_A \subseteq C_V \subseteq C_T$$

(that is, all actuator cells are voted, but no sensor cells are).[5]

Each execution step in the replicated machine takes place in two stages. In the first stage, each working component machine performs a single (ordinary) step. This is specified by the function

- $\mathit{sstep}: \mathcal{R} \times C \rightarrow \mathcal{R}$

where

$$\neg\mathcal{F}(i)(\mathit{when}(c)) \supset \mathit{sstep}(\rho, c)(i) = \mathit{step}(\rho(i), c).$$

This definition states that a working component machine updates its own state in exactly the same way the unreplicated system model would, given the same state. Two important consequences of this definition may not be obvious:

- If cell c is a sensor cell, then the value of $\mathit{step}(\rho(i), c)$ is

  $$\rho(i) \textbf{ with } [c := \mathit{sensor}(c)(\mathit{when}(c))]$$

  (this comes from the definition of *step*). Note that the expression in the **with** clause is independent of the machine i; thus, as noted above, our model requires that all working machines get exactly the same sensor samples.

- If machine i is failed when execution of cell c should be performed, we know nothing whatsoever about the subsequent state of that machine, i.e., $\mathit{sstep}(\rho, c)(i)$. We do *not* assume merely that the value stored for cell c could be incorrect; we allow the whole state (of that machine) to be damaged or destroyed.

When a voted cell is executed, the working component machines each calculate the majority vote of the full set of all their individual results. This is specified by the function

[5] Sensor cells are not voted because we assume an underlying Byzantine fault-tolerant distribution mechanism which ensures that all working machines get the same sensor samples. This assumption is captured in the definition of the function *sstep*.

- $vote: \mathcal{R} \times C \rightarrow \mathcal{R}$

where

$$\neg\mathcal{F}(i)(when(c)) \supset vote(\rho, c)(i) = \rho(i) \textbf{ with } [c := maj\{\!\{\rho(j)(c) | j \in R\}\!\}],$$

maj is the "majority" function, and $\{\!\{\rho(j)(c) | j \in R\}\!\}$ denotes the bag (multiset) of values recorded for cell c by all the component machines.[6]

As with the *sstep* function, we know absolutely nothing about the state of a failed component machine after a vote in which it should have participated. Another interesting element of this definition is that all working machines are specified to perform a majority vote on the *same* bag of values: this suggests they must not only read each other's values correctly, but they should agree on the values attributed to faulty components. These are precisely the requirements that "Byzantine agreement" (also known as "interactive consistency") algorithms are required to satisfy. It may seem, therefore, that any realization of this model should employ a Byzantine agreement algorithm to distribute the values to be voted among all of the component machines. This is unnecessary, however, since it is a majority vote that is being computed, and our results will establish that the good values comprise a majority. Thus, the values ascribed to failed processors are irrelevant, and the working processors do not, in fact, need to agree on those values. We do not prove this result here; we regard it as a proof obligation on the implementation.

The overall behavior of the replicated machine is specified by the function

- $rstep: \mathcal{R} \times C \rightarrow \mathcal{R}$

which is simply the appropriate combination of the two steps above:

$$rstep(\rho, c) \stackrel{\text{def}}{=} \begin{cases} vote(sstep(\rho, c), c) & \text{if } c \in C_V \\ sstep(\rho, c) & \text{otherwise.} \end{cases}$$

Functions *rrun* and *rrunto* are defined analogously to the single machine case:[7]

- $rrun: M \rightarrow \mathcal{R}$,

[6] Note that maj is a *partial* function: it is undefined if an absolute majority of components do not agree on a value. Our results will always take care to establish conditions in which it is defined. A fast and very clever algorithm for calculating the majority function was discovered by Boyer and Moore during the SIFT project [57].

[7] Readers unfamiliar with higher-order logic may find the, so-called "Curried," functions that we employ somewhat strange. Rather than the Curried application $rrun(m)(i)(c)$, they might prefer the application of a function with multiple arguments: $rrun(m, i, c)$. The advantage of our approach is that the separate components of the application have individual meaning and can be manipulated individually: $rrun(m)$ is the state of the replicated machine after m steps, $rrun(m)(i)$ is the state of the i'th component machine at that point, and $rrun(m)(i)(c)$ is the value stored for cell c in that state.

is given by

$$\begin{aligned} rrun(0) &\stackrel{\text{def}}{=} (\lambda i\colon run(0)) \\ rrun(m+1) &\stackrel{\text{def}}{=} rstep(rrun(m), sched(m+1)) \end{aligned}$$

and

- $rrunto\colon C \rightarrow \mathcal{R}$

by

$$rrunto(c) \stackrel{\text{def}}{=} rrun(when(c)).$$

Notice that our model assumes that computation and voting are atomic, and that the components of the replicated machine are completely synchronous. These are idealizations of reality and we intend to explore more realistic assumptions in later work. They are adequate, however, for the purpose of the current investigation, where we are primarily concerned to develop the conditions under which majority voting successfully masks transient failures.

2.3 Fault Tolerance and Transient-Recovery

Our goal in this section is to show that, under certain conditions concerning the failure "pattern" $\mathcal{F}$, the replicated machine produces the same actuator behavior as the single machine, despite failures among the components of the replicated machine. Our requirements are that the majority-voted value for each actuator should be the correct value—that is, the value produced by a single fault-free system. In our model, actuator cells are voted, so that any nonfaulty component machine will set its own value for an actuator cell to that of the majority. Thus, the correctness statement can be rephrased as the requirement that the value computed for an actuator cell by any nonfaulty component machine should be the correct value.

We can state the condition that a component machine i have the correct value for cell c in terms of a predicate:

- $good\text{-}value\colon R \times C \rightarrow \{T, F\}$

where

$$good\text{-}value(i, c) \stackrel{\text{def}}{=} rrunto(c)(i)(c) = runto(c)(c).$$

We then seek a predicate

- $safe\colon C \rightarrow \{T, F\}$

such that

$$\forall c \in C_A, i \in R : (\mathit{safe}(c) \wedge \neg\mathcal{F}(i)(\mathit{when}(c))) \supset \mathit{good\text{-}value}(i, c).$$

Intuitively, *safe*(*c*) will capture the conditions under which the replicated machine has enough working components, and those components have been working for long enough since their last failure, that good values form a majority and faults will be masked successfully.

If only actuator cells were voted, it would be trivial to derive the required result: *safe*(*c*) would be the condition that a majority of components have been working continuously since the very first cell through the computation and vote of cell *c*. That this condition is sufficient follows from the fact that working component machines given the same inputs produce the same results as each other; failed machines can produce anything (including nothing). Thus, the continuously working machines will agree among themselves at every voting stage and, since they are hypothesized to be in the majority, leave their states unchanged. Since actuator cells are voted, any machine that is working during the vote of an actuator cell will acquire the correct value from this continuously-correct majority.

To see that this condition is necessary, suppose that there has not been a majority of components working continuously since the beginning. Then a majority of machines have failed at some time or other prior to the execution of cell *c*. When they failed, they may have destroyed their system state. Since we are now assuming no votes other than at actuators (and actuators do not provide input to other cells), this corruption may persist even after a failed machine starts working again. Thus a failed machine cannot be guaranteed ever to recover fully. Since these machines are hypothesized to form a majority by the time cell *c* is executed, they could outvote the good machines at that point.

Without intermediate voting of state values, a component machine that suffers a transient failure may never fully recover, since there is no way for it to repair its state data. Intermediate voting can allow this repair to take place, so that the conditions in the predicate *safe* become less Draconian. There are many possible strategies for intermediate voting: we can vote at every cell or only at certain cells, and we can vote the entire state, or just some portions of it, or just the value computed at that cell. Voting more data or voting more often than required can be very expensive, using up resources that could be put to better use. Early DFCS maintained very little state data and it was feasible to vote the entire state every frame. Modern systems maintain much more information and it is necessary to be more sparing in the frequency of voting, and in the quantity of data voted. Obviously there is a trade-off here: voting less frequently, or less data at each vote, may increase the time taken to recover from transients, and thereby reduce the reliability of the system. Clearly, overall reliability depends upon the relationship between the voting strategy, the fault arrival rate, and the dataflow dependencies in the system. We need to encode this relationship as the condition in the predicate *safe*. Intuitively, the condition must ensure that, for every cell, a majority of machines have been working for long enough since their last failure that they have acquired correct values (from sensor samples or votes) for data values that ultimately contribute to the value of cell c, and have computed all intermediate values correctly. Stating this condition formally requires some additional definitions.

We define

- $foundation: C \rightarrow \mathcal{P}(C)$, where $\mathcal{P}$ denotes *powerset*,

recursively as follows:

$$foundation(c) \stackrel{\text{def}}{=} \begin{cases} \{c\} & \text{if } c \in (C_S \cup C_V) \\ \{c\} \cup \bigcup\limits_{(b,c)\in\overline{G}} foundation(b) & \text{otherwise} \end{cases}$$

and

- $support: C \rightarrow \mathcal{P}(C)$

by

$$support(c) \stackrel{\text{def}}{=} \begin{cases} \{c\} \cup \bigcup\limits_{(b,c)\in\overline{G}} foundation(b) & \text{if } c \in C_V \\ foundation(c) & \text{otherwise.} \end{cases}$$

The *foundation* of a cell c consists of all those cells that directly or indirectly contribute input data to c by a path that does not pass through any (other) voted cells. Note that a voted or sensor cell is its own foundation.

Figure 2.1 gives a graphical representation of these concepts. In the figure, circles indicate cells, double circles indicate voted cells and the arrows indicate dataflow dependencies (the arrow from cell D to cell A represents the arc $(A, D) \in \overline{G}$; the direction of the arrowhead indicates the dependency relation, rather than the flow of data). The left to right position of cells on the page suggests the order in which they are executed. In this case, the foundation for cell J is just $\{J\}$ (since J is a voted cell), that for A is $\{A\}$ (since A is a sensor cell), and that for cell D is $\{A, C, D\}$.

The *support* for a nonvoted cell is simply the foundation for that cell; the support for a voted cell is the union of the foundations of all the cells that directly provide input to that cell. The intuition here is that if a machine computes correct values for all the cells in *support*(c), and if the machine keeps working, then the value eventually computed for cell c will be correct. In Figure 2.1, the supports for A and D equal their foundations, whereas the support for the voted cell J is $\{A, C, D, J\}$. A machine that is working throughout the support of cell J will compute the correct value for that cell: since it is working at sensor cell A, it will acquire the correct

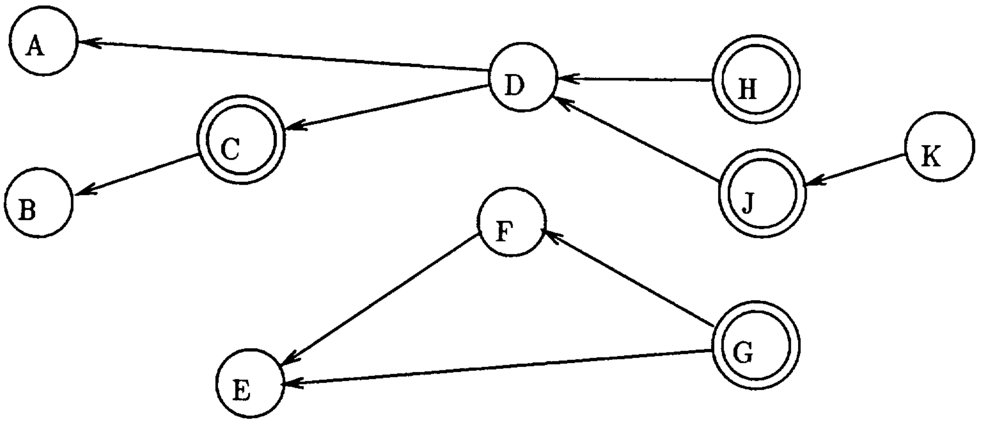


Figure 2.1: Example Dataflow Dependency Graph

sample value from that sensor; since it is working at voted cell C, it will acquire the correct value for that cell during its majority vote, even if it had been failed earlier and had not computed the right value itself [8]; since it has the correct input values for cell D and is working at that cell, it will compute the correct output value; and since it has (from D) the correct input value for cell J, and is working at J, it will compute the correct value for J.

We need just a few more definitions. The function

- *committed-to*: $C \rightarrow M$

is defined by

$$\mathit{committed\text{-}to}(c) \stackrel{\text{def}}{=} \mathit{min}\{\mathit{when}(a) | a \in \mathit{support}(c)\}.$$

In the example of Figure 2.1, $\mathit{committed\text{-}to}(J) = \mathit{when}(A)$. Once a machine reaches $\mathit{committed\text{-}to}(c)$ in its schedule, it must keep working until $\mathit{when}(c)$ if it is to compute the correct value for cell c. Conversely, if it does keep working throughout this period, it will compute the correct value for cell c even if its own state data are corrupt at the beginning of the period. This is because all the data required to compute cell c are derived either from sensor samples, or from voted values, that are acquired at or later than $\mathit{committed\text{-}to}(c)$. Thus, provided enough other machines

[8] We are assuming here that enough machines were working correctly at c that correct values form the majority. We cannot give a characterization of the necessary condition yet, since we are in the process of developing the concepts that make its statement possible.

are working, this machine will acquire good values during the votes and sensor-samples and its own bad state data will not contribute to the result.

The function OK captures the condition under which a particular component machine has been working for "long enough" since its last fault that any bad state data values have been replaced by good values through votes and sensor samples—so that it is able to compute a good result for the current cell. Thus,

- $OK\colon R \rightarrow (C \rightarrow \{T, F\})$

is defined by

$$OK(i)(c) \overset{\text{def}}{=} (\forall m : \textit{committed-to}(c) \leq m \leq \textit{when}(c) \supset \neg\mathcal{F}(i)(m)).$$

In other words, $OK(i)(c)$ is the condition which ensures that component machine i has no state data faults that can affect the value computed for cell c.

For the replicated machine to be *safe*, a majority of its components must be OK for every cell. We therefore introduce the function

- $MOK\colon C \rightarrow \{T, F\}$

(for Majority OK) defined as follows

$$MOK(c) \overset{\text{def}}{=} \exists\Theta \subseteq R, |\Theta| > r/2 : i \in \Theta \supset OK(i)(c).$$

We then define the predicate *safe* as follows

$$\textit{safe}(c) \overset{\text{def}}{=} (\forall a : \textit{when}(a) \leq \textit{when}(c) \supset MOK(a)).$$

That is, the replicated machine is *safe* at cell c if, the condition MOK holds at c itself and at all cells evaluated earlier than c.

Now we can state and prove our main theorem. This "Consensus Theorem" is similar to lemmas of that name in [9].

Theorem 1 (Consensus Theorem) If *safe*(c), then

$$\forall j \in R : OK(j)(c) \supset \textit{good-value}(j, c).$$

Proof: The proof is by strong induction on $\textit{when}(c)$. The basis is the case $\textit{when}(c) = 1$, in which case c must be a sensor cell, and so

$$\textit{rrunto}(c)(j)(c) = \textit{sensor}(c)(1) = \textit{runto}(c)(c)$$

as required.

For the inductive step, suppose the theorem true for all cells a such that $\textit{when}(a) < \textit{when}(c)$ and let j be a component machine such that $OK(j)(c)$. If

$c \in C_S$, the argument is the same as for the basis case, and so we consider $c \in C_T$ and consider a such that $(a,c) \in \overline{G}$. Since the result of c is a function of its inputs, the result will follow if we can demonstrate

$$\textit{good-value}(j,a).$$

There are two cases to consider.

Case 1: $a \in C_V$. It may not be that $OK(j)(a)$ and so we cannot appeal to the inductive hypothesis directly, but we do know that $MOK(a)$ and hence that a majority of machines exemplified by k (possibly not including j) satisfy $OK(k)(a)$. By the inductive hypothesis, *good-value*(k,a) for these machines. Now, we hypothesized $OK(j)(c)$ and hence $\neg\mathcal{F}(j)(a)$. It follows that during the voting stage of the execution of cell a, machine j will acquire the majority value for that cell, i.e., *good-value*(j,a), as required.

Case 2: $a \notin C_V$. A component machine i is OK for cell c if it is working throughout the period from *committed-to*(c) to $when(c)$. Observe that the support of a nonvoted cell a is a subset of any cell c to which it provides input. It follows that *committed-to*(a) can be no earlier than *committed-to*(c). We must also have $when(a) < when(c)$. Thus $OK(i)(c) \supset OK(i)(a)$ and the result then follows directly from the inductive hypothesis.

□

The result we seek follows from the Consensus Theorem:

Corollary 1 For $c \in C_A$, if *safe*(c) then

$$\forall i \in R : \neg\mathcal{F}(i)(when(c)) \supset \textit{good-value}(i,c).$$

Proof: The statement of the corollary implies $MOK(c)$, so there must exist $j \in R$ such that $OK(j)(c)$. The Consensus Theorem then supplies

$$\forall j \in R : OK(j)(c) \supset \textit{good-value}(j,c)$$

which, on expanding the definition of *good-value*, gives

$$rrunto(c)(j)(c) = runto(c)(c).$$

Now $c \in C_A$, so c is a voted cell, and the definition of the voting function ensures,

$$\forall i,j \in R : (\neg\mathcal{F}(i)(when(c)) \wedge \neg\mathcal{F}(j)(when(c))) \supset rrunto(c)(i)(c) = rrunto(c)(j)(c),$$

since all working machines acquire the majority value as the result of voted cells. By definition, $OK(j)(c) \supset \neg\mathcal{F}(j)(when(c))$. Hence, for any $i \in R$ such that $\neg\mathcal{F}(i)(c)$,

$$rrunto(c)(i)(c) = rrunto(c)(j)(c) = runto(c)(c)$$

and we conclude *good-value*(i, c) as required. □

In words, the corollary states that each working component of the replicated machine computes the correct value for an actuator if a majority of machines is working throughout the period from *committed-to*(c) to $when(c)$ for each cell c in the schedule up to and including the actuator concerned.

In Chapter 3, we consider the formal specification of this model in EHDM, and the mechanically-checked verification of the results derived above.

Chapter 3

Specification and Verification in EHDM

In this chapter we give an overview of the formal specification and verification in EHDM of the model presented in Chapter 2. It is not our purpose to provide a general introduction to EHDM here; readers unfamiliar with the EHDM language and system are referred to von Henke and Rushby [7]. Our purpose is rather to discuss some of the more interesting issues raised by the formalization, and to provide a road map to the complete listings of the EHDM specification and verification, which are given in the Appendices. The LaTeX-printed EHDM specification is given in Appendix A; a cross-reference from identifiers to the module in which they are declared is given in Appendix B; Appendix C reproduces the summary from the EHDM proof-chain analysis for the result corresponding to Corollary 2. All the material in the Appendices was generated directly by the EHDM system.

Since the specification language of EHDM is a rather rich, strongly typed higher-order logic, it was possible to cast the model presented in the previous chapter into EHDM fairly directly. The specification of the basic process-control model is given in the module `simple_machine` (page 59). The semantic subtypes of EHDM allowed us to specify the various types of cell in a very natural and convenient manner. For example, C_T, the type corresponding to the active-task cells is specified as the subtype of C (the type of all cells) satisfying $(\lambda c : \mathit{cell\_type}(c) \neq \mathit{sensor\_cell})$. We can then define the signature of the function *task* as $C_T \rightarrow$ task_fn and the EHDM system will ensure that applications of the form $\mathit{task}(c)$ occur only in contexts where c can be proven to satisfy the subtype predicate for C_T. These proof obligations are called *type-correctness-conditions* (or *tcc*'s for short) and are placed in system-generated modules whose names end in `_tcc`. The definition of the function `step`, for example, causes two such tcc's to be generated in the module `simple_machine_tcc` (page 62). This latter module contains several other tcc's, including three that are required to demonstrate the nonemptiness of the subtypes introduced, one that is necessary to demonstrate the well-foundedness of the recursive definition for the function run,[1] and two others that are similar to those just discussed for step. EHDM provides a tool called the *proof-chain analyzer* that checks whether a verification is complete. Among the conditions that it enforces is the requirement that all tcc's be proven.

System-generated tcc modules automatically include trivial proof declarations for the formulas concerned. When these automatically generated proof declarations do not suffice to establish their corresponding theorem, the user must construct more elaborate proof declarations in another module. (Being system generated, and crucial to the type-correctness of the specification, tcc modules are protected against modification by the user.) The three such declarations needed in this case are given in the module simple_machine_tcc_proofs (page 64). A similar naming convention is applied to other modules containing proofs for tcc's. In order to satisfy the nonemptiness requirements on subtypes, we introduce three constants corresponding to an arbitrary sensor, actuator, and active-task cell respectively (strictly, the last of these is unnecessary—actuators are also active tasks). In any application of the specification, instantiations for these constants must be supplied.

We do not define the relation G in the EHDM specification; the simpler relation $\overline{G}$ is sufficient to state and derive all the results required. We introduce initial_state as an arbitrary constant of type state to serve as the initial value in the recursive definition for the function run.

The rest of the specification in module simple_machine is a fairly direct transliteration of that given in Section 2.1, with one exception: the EHDM specification has an extra argument for the function step. This was intended to allow for the description of systems with a less rigid scheduling model than that eventually employed. Thus, whereas Section 2.1 has

$$step(\sigma, c) = \sigma \textbf{ with } [c := \textbf{if } c \in C_S \textbf{ then } sensor(c)(when(c)) \textbf{ else } task(c)(\sigma)],$$

the EHDM specification has

$$step(\sigma, c, m) = \sigma \textbf{ with } [c := \textbf{if } c \in C_S \textbf{ then } sensor(c)(m) \textbf{ else } task(c)(\sigma)].$$

However, this latter version of the function is always used in the form $step(\sigma, c, when(c))$, so that it is equivalent to the first version.

The module simple_props (page 68) states and proves some simple consequences of the previous definitions that are needed later. One, stay_correct_simple, is an example of the type of condition that is often glossed over in conventional mathematical presentations, such as that in Chapter 2. It states that if the output of cell

[1] The annotation "...by identity" in the recursive definition of run establishes identity as the *measure function* for the recursion. The value of the measure function is required to be strictly decreasing across recursive calls, and a tcc is generated to ensure that this is so.

a is used as an input to cell c, then the value recorded for a immediately after it is computed will still be the same when it is accessed (possible much later) in order to be used in the computation of c. In the case of the simple machine, this result is straightforward; it is less so in the case of the replicated machine (since failures must be accounted for). In either case, this is the step that will require a modified proof if the specification is adjusted to model systems that do not keep all cell values for all time (see page 47).

The proof of stay_correct_simple is by induction. The particular form of induction used is a variant of simple induction over the natural numbers. This is stated as the higher-order theorem induction_m in the module natinduction (page 66). This module states two other induction schemes; all three are derived from a statement of Noetherian induction given by the axiom general_induction in the module noetherian (page 65). Note that general_induction is the only induction scheme stated as an axiom; all the others are theorems derived from this single axiom. Notice, too, that the module noetherian has assumptions (stated in the assuming clause) that must be discharged in any instantiation. The module natinduction discharges these assumptions for its particular instantiation.

The next three modules, sets (page 71), cardinality (page 72), and orderedsets (page 74), introduce concepts related to sets that are needed in order to state the model for the replicated machine. Sets are modeled by their characteristic predicates; the type of (the predicate representing) a given set is dependent on the type supplied as the actual parameter to the sets module. The sets module defines the basic set operations of union, intersection, subset, and the like, as higher-order functions. Those unfamiliar with the use of higher-order logic in specifications may find these definitions particularly interesting.

The module cardinality introduces the notion of the cardinality (size) of a set and defines some of its properties axiomatically. Some of the axioms we use, for example

$$|a \cup b| + |a \cap b| = |a| + |b|,$$

are valid only for finite sets. Accordingly, an assumption is attached to this module to ensure that only finite types may be supplied as its actual parameter. The EHDM proof-chain analyzer checks that module assumptions are discharged in any instantiations before the overall verification is declared complete.

The module orderedsets defines the function min (the value of the smallest element) on sets whose elements are drawn from a type with a suitable ordering relation.

The replicated system model is developed in the module repl_machine (page 75). The specification follows very closely that given in Section 2.2. As with the step function of simple_machine, the functions vote, sstep, and rstep all take a third argument in the EHDM specification, but are always used in a manner that is consistent with the two-argument forms given earlier. Another slight difference is in the specification of the condition that majority voting is performed only for voted cells. In the EHDM specification, this is given in the axiom for the vote function, rather than in the definition of rstep. The two approaches are obviously equivalent, but if we were to revise the EHDM specification, we would change it to the alternate form used in Section 2.2. The form currently employed suggests that the voter is always applied, but only actually does a vote when the cell is a voted one; it would be more natural to specify that the voter always votes, but is applied only when the cell is a voted one.

The required property of the maj (majority vote) function is specified in the axiom maj_ax. Note that by specifying this function relative to a set of component machines, rather than relative to the values recorded by their states, we avoid the need to introduce the concept of a multiset. The majority vote function used in Section 2.2 is a partial function: it is undefined if an absolute majority does not exist. Functions in EHDM are total, however: the maj function, for example, has *some* value even when an absolute majority does not exist—we simply know nothing about what that value may be. In order to make use of maj_ax, the verification must always establish that the conditions for the existence of an absolute majority are satisfied. Thus the distinction between a truly partial function and a total function whose values are unconstrained when applied outside its domain is moot in this case.[2]

Module supports (page 79) introduces the functions foundation, support, and committed_to that are needed in the statement of the Consistency Theorem. Subsidiary functions backup and critical_times are used in the definitions.

The module correctness (page 84) defines the functions OK and MOK, the predicates safe and correct, and states the_result, which corresponds to the Consensus Theorem, and is the main result proved in the verification. The definition for safe given in the EHDM specification is weaker than that given in Section 2.3, and so the_result is stronger than Theorem 1 of Section 2.3. The difference is that the formal specification of *safe(c)* requires only that the replicated machine be MOK for those cells a that transitively contribute input to c; the definition in Section 2.3, on the other hand, requires that the replicated machine be MOK for c and for all cells executed earlier than c. Clearly the cells that transitively contribute input to

[2] If $f\colon A \rightarrow B$ is a partial function and $x \in A$ a value outside the domain of definition of f, then the term $f(x)$ has no meaning. There are two ways to capture the useful properties of partial functions in EHDM: one is to use a total function with signature $A \rightarrow B$, but to specify nothing about its values outside its domain of definition. In this case, the term $f(x)$ has some value, but we don't know what it is. Expressions like $x = y \supset f(x) = f(y)$ are meaningful, and true, however.

The other approach is to use a total function with signature $A' \rightarrow B$ where $A' \subset A$ is the true domain. The quotient function, for example, is defined this way in EHDM: quotient: function[number, nznum → number], where *nznum* is the type of nonzero numbers defined as a subtype of numbers by the predicate $(\lambda x\colon x \neq 0)$. In this case, the term quotient(x, y) is type-correct only if it can be proved that $y \neq 0$ in the context of its use.

c must all be executed earlier than c, and so the second condition implies the first. The reason we used a stronger definition for *safe* in the traditional mathematical presentation than we did in the formal specification is that the stronger definition allows Theorem 1 to be proved by simple induction over the natural numbers, whereas the weaker definition requires a proof by Noetherian induction over the structure of the dataflow dependency graph. Noetherian induction is rather tricky to state and carry out in quasi-formal notation (and may not be familiar to all readers) and so we opted for the stronger notion of *safe*, and hence a weaker theorem, in the traditional development. In the truly formal notation of EHDM, it is no more difficult to perform Noetherian than simple induction, and so we used the definition for *safe* that gave the strongest theorem.

The module `connect` (page 87) establishes a crucial lemma called `stay_correct` which states that if a is a cell that provides direct input to cell c, and if all component machines that were OK at a computed the correct value for a, and if the replicated machine is safe at c, then all component machines that are OK for c will have the correct value for a available when they execute c. The proof of this lemma involves a subsidiary lemma called `stay_correct_repl` that is the analog, for the replicated machine, of the `stay_correct_simple` lemma discussed earlier. Like the earlier lemma, this one is proved by induction, but requires a more complex induction scheme than the previous case, because the induction must not proceed beyond the point to which the component machine is known to be OK.[3]

A key step in the proof of `stay_correct` is provided by the lemma `torch_carried`, which establishes that if cell a provides input to cell c, and if the replicated machine is safe at c, then there is some component machine that is OK at both a and c (and hence it "carries the torch" of correct values over from a to c). The proof of this property is the one place where we depend on the fact that we are using *majority* voting (and hence that the intersection of the sets of component machines OK at a and OK at c must be nonempty).

The three modules `sensor_step` (page 91), `nonvoted_step` (page 94), and `voted_step` (page 97) establish the three cases for the inductive step in the proof of `the_result` (i.e., Consensus Theorem) in module `correctness_proof` (page 103). Unlike the traditional-style proof for the Consensus Theorem given in Section 2.3, where strong induction over the schedule of cell executions is employed, the verification in EHDM uses Noetherian induction on the dependency structure recorded in the relation $\overline{G}$. This is the most natural induction scheme to employ in this case and, as noted earlier, allows a stronger formulation of the theorem. Since the statement of the Consensus Theorem has the form of an implication, we actually

[3] The type C of cells is implicitly defined to be infinite by the module `simple_machine`, since the when and sched functions constrain it to be bijective with the naturals. The specification would be improved if the bijection were established with a finite initial subset of the naturals. In this case, the inductive proof of `stay_correct_simple` would also require revision.

employ a specialized form of Noetherian induction called mod_induction that is tailored to this case. The statement and proof of mod_induction appear in the module noetherian.

The three modules concerned with the establishing the inductive step for the proof of the_result each prove a lemma which states, for the case of the cell c considered (i.e., a sensor cell, a nonvoted active-task cell, and a voted cell, respectively), that if the replicated machine is safe at c, and correct at all cells a that provide input to c, then the replicated machine will be correct at c. The proofs of these results essentially follow from applications of the definitions of the functions step, sstep, vote, rstep, rrun, and rrunto, but are somewhat tedious in EHDM since its theorem prover lacks a rewriter: numerous lemmas are required to break the proof down into manageable pieces, each involving the application of just one or two definitions.

Finally, the module outputs contains the specification and proof for the formula actuators_correct, which corresponds to Corollary 1 in Section 2.3.

The complete verification of the_result requires the mechanized checking of 93 proofs (in addition, there are 9 automatically generated tcc proofs that fail; these are supplanted by successful proofs among the 93) and takes about 7 minutes on a Sun SPARCstation 2. The terse proof-chain analysis for the_result is given in Appendix C. The effort required to formally specify and verify the model in EHDM was between three and four man-weeks.

Chapter 4

Reconciliation with the LaRC Model

In this chapter we explain the connection between our model and that developed by Di Vito, Butler and Caldwell of NASA Langley Research Center (LaRC) [9]; for brevity, we will generally refer to this as the "LaRC model."

A major difference between our model and the LaRC model is that we allocate the elementary units of activity to a single-level structure of cells, whereas the LaRC model considers a hierarchy of subframes, frames and cycles (in ascending order). Thus, in our model, cells are drawn from a simple type C, whereas in the LaRC model the units of activity (which we will call "LaRC-cells") are represented by triples which we write as $[p, f, s]$, where p is the cycle,[1] f is the frame, and s is the subframe. There are an indefinite number of cycles, M frames, and frame f has M_f subframes. If we let $\mathbf{N}_k$ denote the first k natural numbers, then we require $p \in \mathbf{N}, f \in \mathbf{N}_M$, and $s \in \mathbf{N}_{M_f}$.[2] The sequence of frames repeats to form cycles; hence the properties of the LaRC model are primarily specified in terms of the last two components of the LaRC triples. Dataflow dependencies are represented by a relation $\rightarrow$ on these pairs, where

$$[f, s] \rightarrow [g, t]$$

means that subframe s of frame f supplies input to subframe t of frame g. If

$$f > g \vee (f = g \wedge s \geq t)$$

[1] We use the variable p, suggesting period, rather than c, suggesting cycle, to avoid confusion with c as a cell.

[2] This is an example of a *dependent* type: a type that depends on the *value* of a variable. EHDM has dependent typing, but lacks a syntax for stating the product type required here. A more advanced specification language under development at SRI permits this type definition to be stated directly.

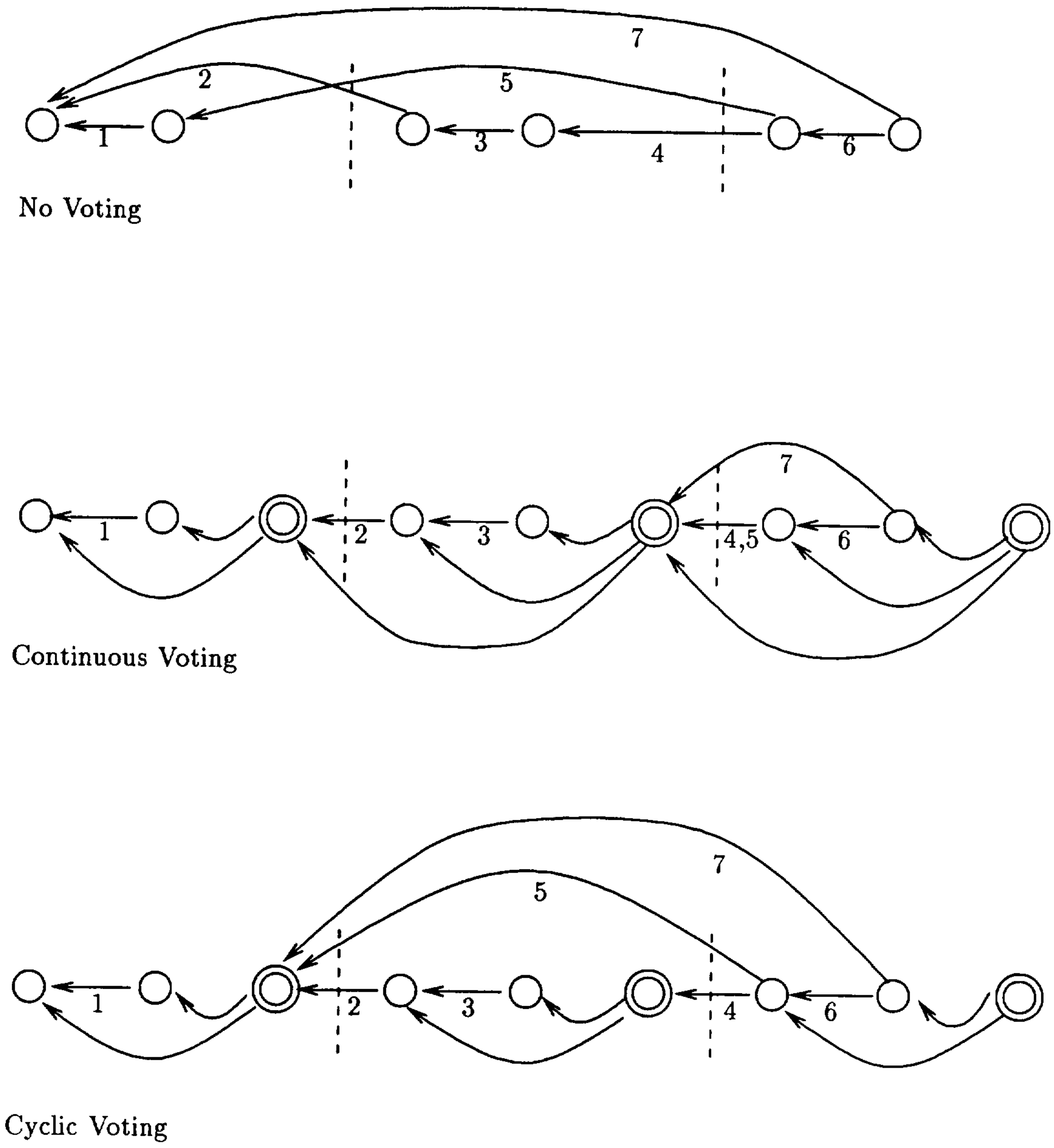
then the input comes from $[f, s]$'s execution in the previous cycle. The directed graph associated with $\rightarrow$ is called the ***task graph***.

The second major difference between the LaRC model and ours is that we associate voting with individual cells, whereas the LaRC model treats voting as a separate activity performed at the end of each complete frame. The LaRC model employs a predicate VP (for Voting Pattern) to indicate what results are to be voted in each frame: $VP(f, s, g)$ is true just in case the result of subframe s in frame f is voted at the end of frame g.

The association of votes with frames in the LaRC model renders it strictly weaker than our model: we can model any system that can be represented within the LaRC model, but we can also model systems (for example, those having votes elsewhere than at the end of the frame) that cannot be represented within the LaRC model. In order to substantiate the first of these claims (the second is self-evident), we now indicate how the LaRC model can be represented within our formulation.

To do this, we introduce a new "voting" cell at the end of every frame in the LaRC task graph and, to a first approximation, we add an arc to the task graph between each (regular) cell and the voting cell of the frame that votes that cell's value; we also replace those dataflow references to the value of the original cell made by cells scheduled in frames later than one that votes its value by references to the value of the voting cell. We say "in principle" because the process is complicated when a value is voted by more than one frame. In this case, the voting cells of the later frames vote on the previously voted value, not on the value of the original cell; similarly, any references to the value always retrieve the most recently voted version. (This is because there really is only one copy of the value).

Figure 4.1 gives a pictorial representation of the transformation just described. In the figure, vertical dashed lines indicate frame boundaries, and the left to right order of cells on the page suggests their temporal order of execution. The top image portrays an unvoted system with three frames and two subtasks in each frame; the numbered arcs indicate the dataflow dependencies. The lower two images portray the system after transformation to frame-based voting systems. The double circles represent the new voting tasks and the unnumbered arcs that curve below the line of circles represent the dataflow dependencies of these new voting tasks. The middle image portrays "continuous voting" (see Section 4.1.1), in which all data are voted every frame—hence each voting task has a link back to the previous voting task in order to access the previously voted values of earlier tasks. Arcs corresponding to the original dataflow references retain the same numbering scheme in this transformed portrayal. Observe that arc number 7, for example, no longer reaches back to a task several frames earlier, but only to the previous voting task. The bottom image portrays the system after transformation to a frame-based voting system using "cyclic voting" (see Section 4.1.2), in which each frame votes only the data generated in that frame. Here, arc 7 must still reach back to the frame containing



No Voting

Continuous Voting

Cyclic Voting

Figure 4.1: Representation of Frame-Based Voting

the task of interest, but the data is acquired from the voting task of the frame concerned.

Formal description of the transformation is complicated by the need to take care of the details. We identify the cells of our model with the triples $[p, f, s]$ of the LaRC task graph, together with an initialization cell and the special voting cells; we denote the initialization cell by c_I, and the voting cell at the end of frame g of cycle q by $v_{(q,g)}$. The basic dataflow connections $\rightarrow$ of the LaRC task graph give rise to edges in our graph $\overline{G}$ as follows:

$$([p,f,s],[q,g,t]) \in \overline{G} \textbf{ iff } [f,s] \rightarrow [g,t] \text{ and}$$

$$\begin{array}{l} p = q \wedge \begin{cases} f < g \\ \quad \vee \ (f = g \wedge s < t) \end{cases} \\ \vee \\ p = q - 1 \wedge \begin{cases} f > g \\ \quad \vee \ (f = g \wedge s \geq t) \end{cases} \end{array}$$

Cells that would otherwise be dependent on frame -1 instead make reference to the initialization cell:

$$(c_I, [0,g,t]) \in \overline{G} \textbf{ iff } [f,s] \rightarrow [g,t] \wedge (f > g \vee (f = g \wedge s \geq t)).$$

The execution schedule for the LaRC model is implicit in the frame structure: all the subframes for frame 0 are executed in order, then those for frame 1, and so until the last subframe of frame $M - 1$, at which point a new cycle starts over at subframe 0 of frame 0. If we let $K(f) = \sum_{g=0}^{f-1} M_g$ denote the total number of subframes in the first f frames of the task graph, then we require

$$when(c_I) = 0,$$

$$when([p,f,s]) = p \times (K(M) + M) + (K(f) + f) + s + 1$$

and

$$when(v_{(q,g)}) = q \times (K(M) + M) + (K(g) + g) + M_g + 1.$$

We define orderings $>$ and $\geq$ over $(cycle, frame)$ pairs based on their position in the execution sequence:

$$(p,f) > (q,g) \textbf{ if } (p > q) \vee (p = q \wedge f > g), \text{ and}$$

$$(p,f) \geq (q,g) \textbf{ if } (p > q) \vee (p = q \wedge f \geq g).$$

We also use the inverse relations $<$ and $\leq$ whenever convenient and extend the relations to voted cells by the convention

$$v_{(q,g)} \leq v_{(r,h)} \textbf{ iff } (q,g) \leq (r,h).$$

A voting cell $v_{(q,g)}$ is a *candidate* voting cell for ordinary cell $[p, f, s]$ if $VP(f, s, g)$, and either $q = p \wedge g \geq f$ or $q = p + 1 \wedge g < f$; the candidate cell that is least with respect to the $\leq$ ordering is the *primary* voting cell for $[p, f, s]$, the others are *secondary* voting cells for $[p, f, s]$.

An arc $([p, f, s], v_{(q,g)})$ is added to $\overline{G}$ when $v_{(q,g)}$ is the primary voting cell for $[p, f, s]$. An arc $(v_{(q,g)}, v_{(r,h)})$ is added to $\overline{G}$ when $v_{(r,h)}$ is a secondary voting cell for $[p, f, s]$, and $v_{(q,g)}$ is the largest candidate voting cell for $[p, f, s]$ with respect to the $\geq$ ordering such that $(q, g) < (r, h)$. Finally, we replace arcs $([p, f, s], [q, g, t]) \in \overline{G}$, by arcs $(v_{(r,h)}, [q, g, t])$ where $v_{(r,h)}$ is the largest candidate voting cell for $[p, f, s]$ with respect to the $\geq$ ordering such that $(r, h) < (q, g)$.

We claim that the transformation just described will cast an instance of the LaRC model into an instance of our model in a way that preserves its essential properties. Despite its notational complexity, the transformation is really quite simple: it "unrolls" the cyclic schedule of the LaRC model into flat structure that we require, and it encodes the frame-based voting of the LaRC model in the voted cells of our model.

4.1 Specific Voting Patterns

In the following sections we will derive results similar to those of [9, Section 14] for specific voting patterns. We will use the general character of the transformation between the LaRC model and ours described above, but will not undertake literal translations of the LaRC Theorems. Instead, we will state what we consider to be the main thrust of the LaRC Theorems directly in the terms of our model, and will conduct our proofs within that context. In this way, we avoid the tedious labor of the transformation, preserve the clarity of the presentation of each result, and increase its generality of application. We claim, but do not prove, that if the statements of the Theorems of [9, Section 14] are transformed in the way described above, then the resulting "mapped" theorems will be special cases of those given below.

All we require to state our first two results is a notion of "frame." The idea is that all cells belong to exactly one frame; the members of each frame are executed sequentially; the last cell executed in each frame is a voted cell, and no other cells are voted.

Thus we introduce the set

- $F = \{0, 1, \ldots, |f|\}$ of *frames*, with mapping
- *frame*: $C \longrightarrow F$, and equivalence relation
- $\sim \subseteq C \times C$

where

$$a \sim c \overset{\text{def}}{=} \mathit{frame}(a) = \mathit{frame}(c).$$

Thus *frame*(c) denotes the frame to which cell c belongs, and $a \sim c$ indicates that a and c both belong to the same frame. The requirement that all the members of a frame are executed in sequence, with no members of other frames intervening, is simply stated by the requirement that the derived function

- *frame-sched*: $M \rightarrow F$,

given by

$$\textit{frame-sched}(m) \stackrel{\text{def}}{=} \textit{frame}(\textit{sched}(m)),$$

should be monotonic increasing.

The final cell executed in a frame is the only voted cell in that frame:

$$C_V \stackrel{\text{def}}{=} \{c | \forall a : a \sim c \supset \textit{when}(a) \leq \textit{when}(c)\}.$$

It is convenient to let *voted-cell*(f) denote the voted cell for frame f.

Equipped with these definitions, we can state and prove results about increasingly less restricted frame-based voting patterns.

4.1.1 Continuous Voting

The idea here is that the entire state of the replicated machine is voted every frame. Thus, any cell that requires a value from an earlier frame need only refer to the voting cell of the immediately preceding frame. Hence, our formalization is:

Definition 1 (Continuous Voting) A replicated machine performs continuous voting if:

$$(a, c) \in \overline{G} \supset a \sim c \vee a = \textit{voted-cell}(\textit{frame}(c) - 1).$$

■

We have

Theorem 2 *If a majority of machines is working throughout each consecutive pair of frames, then the replicated machine is safe under continuous voting.*

Proof: For any cell c, we need to ensure that a majority of component machines are working throughout the period from *committed-to*(c) to *when*(c). The definition of continuous voting ensures

$$\textit{when}(\textit{voted-cell}(\textit{frame}(c) - 1)) \leq \textit{committed-to}(c)$$

and

$$\textit{when}(c) \leq \textit{when}(\textit{voted-cell}(\textit{frame}(c))).$$

Hence, the requirement that a majority of machines are working throughout each consecutive pair of frames is sufficient to ensure that the replicated machine is safe.
□

4.1.2 Cyclic Voting

The idea here is that cells in frame f never refer to cells from frames earlier than $f - e$, where e is a parameter to the design. Further, when cells make "out of frame" references, it is only to voted cells.

Definition 2 (Cyclic Voting) A replicated machine performs cyclic voting with period e if:

$$(a, c) \in \overline{G} \supset a \sim c \vee (a = \textit{voted-cell}(\textit{frame}(c) - k) \wedge 1 \leq k \leq e).$$

(Obviously, there is also a well-formedness condition: $\textit{frame}(c) - k \geq 0$.) Notice that cyclic voting reduces to continuous voting when $e = 1$.

Theorem 3 *If a majority of machines are working throughout each sequence of* $e+1$ *consecutive frames, then the replicated machine is safe under cyclic voting.*

Proof: For any cell c, we need to ensure that a majority of component machines are working throughout the period from $\textit{committed-to}(c)$ to $\textit{when}(c)$. The definition of cyclic voting ensures

$$\textit{when}(\textit{voted-cell}(\textit{frame}(c) - e)) \leq \textit{committed-to}(c)$$

and

$$\textit{when}(c) \leq \textit{when}(\textit{voted-cell}(\textit{frame}(c))).$$

Hence, the requirement that a majority of machines are working throughout each consecutive sequence of $e + 1$ of frames is sufficient to ensure that the replicated machine is safe. □

4.1.3 Optimal Voting

In this section, we examine conditions that allow a replicated machine to vote as little data as possible, and as seldom as possible, yet still be able to recover from transient failures in a fixed amount of time.

The general condition is very simple to state, but not very interesting:

Lemma 1 If there exists a constant B such that

$$\forall c : \textit{when}(\textit{voted-cell}(\textit{frame}(c) - B)) \leq \textit{committed-to}(c),$$

and a majority of machines are working throughout each sequence of $B + 1$ consecutive frames, then the replicated machine is safe.

Proof: This result follows by the same argument used in Theorem 3. □

The conditions become more interesting when we consider cyclic schedules. It is natural and convenient to think of cyclic schedules as generated by repeatedly "unrolling" a more basic schedule for a single cycle. We assume such basic schedules to be composed of "basic cells" of the form $[f, s]$ where f is the *frame*, and s the *subframe*. A relation $\rightarrow$ defines the dataflow relationships among the basic cells: $[f, s] \rightarrow [g, t]$ means that subframe s of frame g provides input to subframe t of frame g. Cells are executed in order by frames, and in subframe order within frames. As before, we assume there are M frames.

So far, this model is the same as the LaRC model [9]; a difference is that here we allow arbitrary basic cells to be designated as voted cells, whereas the LaRC model considers voting to take place at the end of each frame and indicates that cell $[f, s]$ is voted in frame n by $VP(f, s, n)$. As explained at the beginning of this chapter, there is a straightforward transformation from the standard LaRC model to the variant used here.

The *frame length* of a step $[f, s] \rightarrow [g, t]$ is defined by

$$\begin{array}{rll} 0 & \textbf{if} & f = g \wedge s < t, \\ M & \textbf{if} & f = g \wedge s \geq t, \\ g - f & \textbf{if} & f < g, \text{ and} \\ M + (g - f) & \textbf{if} & f > g \end{array}$$

A *path* in the basic schedule is a sequence of cells

$$< [f, s], [g, t], \ldots, [h, u] >$$

such that

$$[f, s] \rightarrow [g, t] \rightarrow \cdots \rightarrow [h, u].$$

The frame length of a path is the sum of the frame lengths of its individual steps.

We "unroll" the basic model to yield cells of the form $[p, f, s]$ where p is the *cycle*, and f and s are the frame and subframe as before. The graph $\overline{G}$ comprises pairs of cells $([p, f, s], [q, g, t])$ such that $[f, s] \rightarrow [g, t]$ in the basic model and

$$\begin{array}{rll} p = q & \textbf{if} & (f < g) \vee (f = g \wedge s < t) \\ p = q - 1 & \textbf{if} & (f > g) \vee (f = g \wedge s \geq t) \end{array}$$

A cell $[p, f, s]$ is voted if $[f, s]$ is designated as a voted cell in the basic schedule; $[p, f, s]$ is a sensor cell if it has indegree zero in $\overline{G}$ (i.e., if there is no basic cell $[g, t]$ such that $[g, t] \rightarrow [f, s]$).

The *frame-time* of a cell is its position in the execution sequence:

$$\textit{frame-time}([p, f, s]) = p \times M + f;$$

the frame length of an arc $([p,f,s],[q,g,t])$ in the graph $\overline{G}$ is defined to be

$$\mathit{frame\text{-}time}([q,g,t]) - \mathit{frame\text{-}time}([p,f,s]).$$

Notice that the construction ensures that this value is nonnegative, and that it equals the frame length of $[f,s] \rightarrow [g,t]$ in the basic schedule. A *path* in the (unrolled) schedule is a sequence of cells

$$< [p,f,s],[q,g,t],\ldots,[r,h,u] >$$

such that each consecutive pair of cells are connected by an arc in the graph $\overline{G}$. The frame length of this path is defined as

$$\mathit{frame\text{-}time}([r,h,u]) - \mathit{frame\text{-}time}([p,f,s]).$$

It is easy to see that this equals the sum of the frame lengths of the individual arcs, and that it also equals the frame length of the basic path

$$< [f,s],[g,t]\ldots[h,u] > .$$

A path

$$< [p,f,s],[q,g,t],\ldots,[r,h,u] >$$

is a *commitment path* if

- The cell [p, f, s] is either a sensor cell or a voted cell, and
- No other cells in the sequence are voted, except possibly the last.

Then we have

Lemma 2 If there exists a bound B on the frame-length of any commitment-path, and a majority of machines are working throughout each sequence of $B+1$ consecutive frames, then the replicated machine is safe.

Proof: If

$$< [p,f,s],[q,g,t],\ldots,[r,h,u] >$$

is a commitment-path, then

$$[p,f,s] \in \mathit{support}([r,h,u]).$$

If no commitment-path has frame length longer than B, it follows that

$$\mathit{when}(\mathit{voted\text{-}cell}(\mathit{frame}([r,h,u] - B))) \leq \mathit{committed\text{-}to}([r,h,u])$$

and the result follows by the previous lemma. □

The existence of the bound B is determined by the presence of vote-free cycles (loops) in the basic task graph:

Lemma 3 There exists a bound B on the frame-length of any commitment-path if and only if all cycles in the basic task graph contain at least one voted cell.

Proof: Suppose there is no such bound B. Then there are commitment-paths of arbitrary frame lengths—and therefore of arbitrary lengths, since the frame length of any individual step is fixed. Since the number of basic cells is fixed and finite, it follows that there must exist a commitment-path of the form

$$< \ldots [p, f, s] \ldots [q, f, s] \ldots >$$

in which the components of some unvoted basic cell $[f, s]$ are repeated and no voted cells appear in between. The construction of the graph $\overline{G}$ is such that this can only happen if there is a cycle

$$[f, s] \rightarrow \cdots \rightarrow [f, s]$$

in the task graph comprising only unvoted cells.

Suppose, on the other hand, that the basic task graph contains a cycle

$$[f, s] \rightarrow \cdots \rightarrow [f, s]$$

comprising only unvoted cells. Then a commitment-path can be constructed containing a segment derived from enough iterations of this basic cycle that the frame-length exceeds any fixed bound B. □

Combining these lemmas, we obtain

Theorem 4 *Recovery from transient faults is possible if and only if there are no vote-free cycles in the basic task graph. Further, if all paths of the form*

$$[f, s] \rightarrow [g, t] \rightarrow \cdots \rightarrow [h, u],$$

where at most the first and last elements are voted, have path lengths no longer than B, and if a majority of machines are working throughout each sequence of $B + 1$ consecutive frames, then the replicated machine is safe.

Proof: Combine the preceding three lemmas. □

Chapter 5

Discussion and Conclusions

We begin with a consideration of possible extensions to this work. These extensions fall into four categories, listed in order of increasing complexity:

- Proof of additional properties within the current model,
- Modification of the current model in order to enhance its abstractness,
- Development of more concrete models on top of the current model, and
- Significant extensions to the model in order to encompass a wider class of systems.

We consider each of these categories in turn.

A topic where additional proofs would expose the underlying requirements more clearly concerns the retention of stored values. The current model treats the system state as a function recording the values of all cells encountered during the entire lifetime of the system. Obviously this is not how we expect the system to be implemented. It is intuitively clear that the only cells whose values need to be retained are those which have been computed but not yet used—that is, the value of cell c needs to be retained only for the interval from $when(c)$ to $max\{when(a)|(c,a) \in \overline{G}\}$.

This can be specified by modifying the definition of the basic function *step*. Currently, we have

$$step(\sigma, c) \overset{\text{def}}{=} \sigma \textbf{ with } [c := \textbf{if } c \in C_S \textbf{ then } sensor(c)(when(c)) \textbf{ else } task(c)(\sigma)].$$

This definition can be replaced by two axioms specifying a modified function *step'*:

$$step'(\sigma, c)(c) = \textbf{if } c \in C_S \textbf{ then } sensor(c)(when(c)) \textbf{ else } task(c)(\sigma)$$

and

$$\forall a, b : (a, b) \in \overline{G} \wedge when(a) < when(c) < when(b) \supset step'(\sigma, c)(a) = \sigma(a).$$

To establish that *step'* is an adequate replacement for *step* we need to prove that the actuator commands are the same in both cases.

There are two ways to carry out this proof. One would establish a variant specification for `simple_machine` using *step'* instead of *step*, and would prove that actuator outputs are the same in both cases—that is, it would verify a theorem of the form $runto(c)(c) = runto'(c)(c)$. This approach would leave the existing specification and verification unchanged but would require a fairly extensive new verification that would mirror, in many respects, the verification already performed. The other approach would modify `repl_machine` to use *step'* instead of *step* and would then carry this additional complication along in the proof of fault masking. This approach is probably the simplest, since the definition of *step* is used only five times in proofs concerning the replicated machine.

A topic where increased abstraction in the current model and verification would expose underlying requirements more clearly is the choice of voting strategy. The current model is firmly based on majority voting, but other strategies such as plurality voting have attractions. As long as the working machines constitute an absolute majority, plurality voting exhibits the same behavior as majority voting. If the working machines should fail to form an absolute majority, however, the majority-voted system will break down, whereas a plurality-voted system may break down or may not, depending on whether enough of the failed machines agree on a common, wrong value to win the plurality vote. There seems to be no way to measure the likelihood of this latter event, nor any sound way to engineer a system so that failed machines are unlikely to agree, and so we do not advocate the use of plurality voting as a way to enhance the claimed reliability of the system. There seems little harm, however, and possibly some value, in using voting strategies that are more robust than strict majority—so that there is at least some chance the system may continue to work even after an explosion, or other catastrophic event, has rendered 10^{-9} irrelevant.[1]

These considerations provide the motivation for a more careful examination of the voting and fault-model assumptions required for the Consensus Theorem to hold. There are two places in the present development where the properties of strict majority voting are employed. One, noted in Chapter 3, is in the proof of `torch_carried`, the other is in the proof of `vote_lemma` in module `voted_step`. It would be very worthwhile to revisit these proofs and to determine a minimal characterization of the properties actually required of the voting function in order for the fault-masking properties to be retained. (Majority is a strict requirement for the `torch_carried` property, but there seem to be other ways to conduct the part of the proof in which this property is used.) The ability to conduct such investigations is one of the benefits of a truly formal development: the axiomatic and definitional

[1] Paul Miner of NASA LaRC first drew these considerations to our attention.

basis of the development is known precisely, and the effect of controlled variations can be rigorously explored.[2]

A prime candidate for a more concrete model to be constructed on top of the one developed here is that of Di Vito, Butler and Caldwell. As indicated in Chapter 4, the main results proved for that model can also be derived from ours; it would be interesting to formally verify those derivations. At a later stage in this program of work, when an actual design for a reliable computing platform for DFCS has been developed, it will be valuable to attempt to instantiate our model for that design.

The characteristics of some potential system designs cannot be seen as instantiations of our model: it will be necessary to significantly revise and extend the model in order to accommodate such designs. Among the revisions and extensions that would be most illuminating are those that break the lock-step synchronization of task executions in the component machines. One extension would still require the same workload for each component machine, but would allow them to execute different schedules. Obviously there are constraints that require a notion of "consistency" to be satisfied among schedules—they must synchronize for votes and must not deadlock, for example. The practical benefit of allowing different schedules on different channels is that simultaneous transient failures of several channels, such as a lightning strike might induce, will be less likely to all affect the activations of a single task; instead, the damage will be shared among several different tasks, and all may still be executed by a majority of working processors.

Another extension would introduce different workloads for different machines. This allows different quantities of replication for different activities and permits better utilization of resources. For example, one really critical activity may run on all processors, another less critical one may run on only three, while another, presumably unimportant, task may run on but a single machine.

So much for future extensions; we now turn to a consideration of the significance of the work actually performed. The work described is just one of the first steps in a much larger program and it would be premature to evaluate the overall program at this stage. We can, however, ask what the model developed here contributes to a science of DFCS design, and we can ask what further value is contributed by its formal specification and verification.

Clearly, our model addresses only a small fragment—redundancy management—of the overall problem of DFCS design, and is a highly abstracted representation of

[2] It may seem moot to explore the circumstances under which a Consensus Theorem can hold with less than $\frac{N+1}{2}$ working channels when the underlying Byzantine fault tolerant sensor distribution and clock synchronization algorithms require $\frac{2N+1}{3}$ working channels. Our response is that it would be worthwhile to investigate the behavior of these Byzantine fault-tolerant algorithms when fewer than the required channels are available. It should be possible to tolerate nonByzantine failures with only $\frac{N+1}{2}$ working channels, but it is unknown whether the standard Byzantine algorithms do so. There has, however, been some investigation of algorithms that tolerate multiple failure modes [58, 59].

that fragment. Small though that fragment may be, however, the evidence cited in Section 1.2 suggests that it is one of the most crucial problems; if managed poorly, redundancy can reduce, rather than enhance, the overall reliability of a DFCS. Recall the summary of Mackall [23, pp. 40–41] quoted on page 8, and which reads in part:

> "...qualification of such a complex system as this, to some given level of reliability, is difficult ...[because] the number of test conditions becomes so large that conventional testing methods would require a decade for completion. The fault-tolerant design can also affect overall system reliability by being made too complex and by adding characteristics which are random in nature, creating an untestable design.
>
> "...reducing complexity appears to be more of an art than a science and requires an experience base not yet available. If the complexity is required, a method to make system designs more understandable, more visible, is needed."

The purpose of the work described here (and of the larger program) is precisely to address these pleas for testable designs, purged of "random characteristics," and which are more "understandable, more visible."

We contend that our model shows that certain principles of design—Byzantine fault tolerant distribution of sensor samples, loosely synchronized execution, majority voting of all actuator outputs, and periodic majority voting of internal state data—provide predictable behavior that masks faults and provides transient-recovery. These principles of design are encoded in the axioms and definitions of our model; the conclusion is derived by mathematical reasoning from that basis.

Other models have been devised that address similar problems. A general method, known rather misleadingly as the "state-machine approach" for constructing reliable systems from unreliable components that periodically vote their results was developed by Lamport in a series of classic papers [60–62] (see also Schneider's tutorial [63]). The development here can be seen as a modification of Lamport's "state-machine" approach to the case where voting is performed intermittently.

The model most similar to our is, of course, that of Di Vito, Butler and Caldwell [10, 9]. The formal connection between the two models was discussed in Chapter 4; here we consider less tangible issues—style, abstractness, and the influence of formal verification.

A maxim usually attributed to Einstein holds that a theory should be "as simple as possible—but no simpler." In our domain, simplicity is closely related to the abstractness of the model considered: the advantage of abstraction is that it reduces a problem to its simplest form and exposes its essential properties to scrutiny, uncluttered by extraneous matter; the danger is that too much is left out, so that the model fails to capture those aspects of reality that are of interest. When formal verification is undertaken, abstraction has economic, as well as philosophical consequences: it will generally be easier, and hence require less resources, to verify an abstract model than a more concrete one. Furthermore, the abstract model should have wider applicability, and hence the cost of its verification can be amortized over more instantiations. Of course, the cost of one instantiation must be borne in order to reach the level of detail considered in the more concrete model.

Our model is considerably more abstract than that of Di Vito, Butler and Caldwell; we explained the reasons for our choices in Section 1.3.1 and considered the reconciliation between the two models in Chapter 4. For the purpose of formal verification, we consider our model to have distinct advantages: it has been subjected, essentially without change, to formal specification and mechanical proof checking in EHDM, whereas we believe that direct verification of the LaRC model would be a considerable challenge. Whether the added concreteness of the LaRC model renders it a more effective specification for human review is something we leave to our readers to decide.

The remaining question we consider is whether formal specification and mechanical proof checking added anything of value to the quasi-formal description and proof presented in Chapter 2. The first thing to note is that the description and proof given in Chapter 2 were heavily influenced by the formal verification—both before and after the latter was performed. It was influenced even before the formal verification was attempted because the model was constructed with formal specification and verification (in EHDM) in mind. Hence, it is expressed directly in terms of (higher-order) functions; the LaRC model, on the other hand, uses vectors, sequences, sets, and iterated conjunction operators. These can all be expressed in terms of (higher-order) functions and we would not hesitate to use them where they contribute to clarity—on the other hand, we generally prefer to do without these constructs when a comparably simple specification can be found that is expressed directly in terms of functions. After the formal verification had been performed, we revised some of the definitions and the proof of Chapter 2 in order to bring them more closely into line with the corresponding EHDM versions.

There is one improvement derived from the formal verification that we did not retrofit to development of Chapter 2: this is a stronger formulation of the main Consensus Theorem. The Consensus Theorem is stated as

If safe(c), *then*

$$\forall j \in R : OK(j)(c) \supset \text{good-value}(j, c).$$

where

$$\text{safe}(c) \stackrel{\text{def}}{=} (\forall a : when(a) \leq when(c) \supset MOK(a)).$$

In the EHDM verification, the Theorem was strengthened by giving a weaker (recursive) definition for *safe*:

$$safe(c) \stackrel{\text{def}}{=} MOK(c) \wedge (\forall a : (a, c) \in \overline{G} \supset safe(a)).$$

The stronger theorem requires only that the replicated machine is MOK for all those cells that transitively contribute input to cell c; the weaker form requires it be MOK for all cells executed prior to c.

Obviously, the stronger theorem could have been stated and proved in the quasi-formal development just as well as the weaker one. The significant point, however, is that it was the weaker formulation, and correspondingly a proof by simple induction, that arose most naturally in the quasi-formal development. In formal verification, the familiar convenience of simple induction is less of a driving force, and we were led to contemplate the stronger theorem, which requires a more difficult Noetherian induction.

The main benefit that we see accruing from the mechanically checked verification is the precision with which the underlying assumptions are now known. Formally, this basis consists of 18 axioms (of which only 11 are directly concerned with the model, while the remaining 7 deal with supporting concepts such as cardinality), and 15 definitions (which provide only conservative extensions in EHDM). Informally, we have acquired a much better appreciation of the issues concerning the retention of stored values, and of the way in which fault masking is dependent on the properties of majority (as opposed to other kinds of) voting. As described above, we are now in a position to investigate these issues formally.

In future work, we hope to explore these issues, and also to extend our formal specification and verification toward the behavior of a realistic operating system that will implement the fault-masking techniques modeled here. The next step will be to combine the model used here with that for clock synchronization [38], in order to consider the more realistic case of replicated computers that are synchronized only within some bound δ, and in which computation and communication take a certain amount of time.

Acknowledgements

We are grateful to Ricky Butler of NASA Langley Research Center for posing the challenge of applying formal methods to aspects of digital flight control systems, and for structuring the overall problem into manageable pieces. Our treatment of the problem tackled in this report owes much to discussions with Ben Di Vito, and to his model for fault masking and transient recovery. Jim Caldwell provided valuable assistance and encouragement in the first stage of the formal verification reported here.

References

[1] Spitzer, Cary R.: *Digital Avionics Systems.* Prentice Hall, Englewood Cliffs, NJ, 1987.

[2] Federal Aviation Administration Technical Center, Atlantic City, NJ. *Digital Systems Validation Handbook–Volume II*, Feb. 1989. DOT/FAA/CT-88/10.

[3] Radio Technical Commission for Aeronautics, Washington, DC. *Software Considerations in Airborne Systems and Equipment Certification*, Mar. 1985. DO-178A.

[4] Computer Science Laboratory, SRI International, Menlo Park, CA. EHDM *Specification and Verification System Version 4.1—User's Guide*, November 1988. See [6] for the updates to Version 5.2.

[5] Computer Science Laboratory, SRI International, Menlo Park, CA. EHDM *Specification and Verification System Version 5.0—Description of the* EHDM *Specification Language*, January 1990. See [6] for the updates to Version 5.2.

[6] Computer Science Laboratory, SRI International, Menlo Park, CA. EHDM *Specification and Verification System Version 5.2—Supplement to User's and Language Manuals*, March 1991. Current version number is 5.2.0.

[7] Rushby, John; von Henke, Friedrich; and Owre, Sam: *An Introduction to Formal Specification and Verification Using* EHDM. Computer Science Laboratory, SRI International, Technical Report SRI-CSL-91-2, Menlo Park, CA, Feb. 1991.

[8] von Henke, Friedrich; Shankar, Natarajan; and Rushby, John: *Formal Semantics of* EHDM. Computer Science Laboratory, SRI International, Menlo Park, CA, January 1990. This document describes EHDM Version 5.0; see [6] for informal descriptions of the changes in Version 5.2.

[9] Di Vito, Ben L.; Butler, Ricky W.; and Caldwell, James L.: *Formal Design and Verification of a Reliable Computing Platform for Real-Time Control.* NASA TM-102716, Oct. 1990.

[10] Di Vito, Ben L.; Butler, Ricky W.; and Caldwell, James L.: High Level Design Proof of a Reliable Computing Platform. In *2nd. International Working Conference on Dependable Computing for Critical Applications*, Tucson, AZ, Feb. 1991, IFIP WG. 10.4, pp. 124–136.

[11] Droste, Carl S.; and Walker, James E.: *The General Dynamics Case Study on the F16 Fly-by-Wire Flight Control System.* AIAA Professional Study Series. American Institute of Aeronautics and Astronautics. Undated.

[12] Learmount, David: A320 Certification: the Quiet Revolution. *Flight International*, February 27, 1988, pp. 21–24.

[13] Anonymous: French Report Details 1988 Crash of A320 following Air Show Flyby. *Aviation Week and Space Technology*, June 4 to July 30, 1990, pp. 107–108, 78–79, 99–103, 98–99, 60–64, 90–93, and 90–93. (Continued over seven issues of the magazine).

[14] Puyplat, Didier: A320: First of the Computer-Age Aircraft. *Aerospace America*, vol. 29, no. 5, May 1991, pp. 28–30.

[15] Dornheim, Michael A.: X-31 Flight Tests to Explore Combat Agility to 70 Deg. AOA. *Aviation Week and Space Technology*, March 11, 1991, pp. 38–41.

[16] Morse, W.D.; and Ossman, K.A.: Model Following Reconfigurable Flight Control System for the AFTI-F16. *AIAA Journal of Guidance, Control, and Dynamics*, vol. 13, no. 6, November-December 1990, pp. 969–976.

[17] Ishmael, Stephen D.; Regenie, Victoria A.; and Mackall, Dale A.: *Design Implications from AFTI/F16 Flight Test.* NASA TM-86026, 1984.

[18] Traverse, P.: Dependability of Digital Computers on Board Airplanes. In *International Working Conference on Dependable Computing for Critical Applications*, Santa Barbara, CA, Aug. 1989, IFIP WG. 10.4, pp. 53–60.

[19] Federal Aviation Administration. *System Design Analysis*, September 7, 1982. Advisory Circular 25.1309-1.

[20] Butler, Ricky W.; and Johnson, Sally C.: *The Art of Fault-tolerant System Reliability Modeling.* NASA TM-102623, Mar. 1990.

[21] Manacher, G.K.: Production and Stabilization of Real-Time Task Schedules. *Journal of the ACM*, vol. 14, no. 3, July 1967, pp. 439–465.

[22] Richards, P.: *Timing Properties of Multiprocessor Systems.* Tech. Operations Inc., Technical Report TDB60-27, Burlington, MA, Aug. 1960.

[23] Mackall, Dale A.: *Development and Flight Test Experiences with a Flight-Crucial Digital Control System.* NASA TP-2857, 1988.

[24] Mackall, Dale A.: AFTI/F-16 Digital Flight Control System Experience. In Beasley, Gary P., editor 1984:, *NASA Aircraft Controls Research 1983.* NASA Conference Publication 2296, 1984, pp. 469–487. Proceedings of workshop held at NASA Langley Research Center, October 25–27, 1983.

[25] Mackall, Dale A.; and Allen, James G.: A Knowledge-Based System Design/Information Tool for Aircraft Flight Control Systems. In *AIAA Computers in Aerospace Conference VII*, Monterey, CA, Oct. 1989, pp. 110–125. Collection of Technical Papers, Part 1.

[26] Kieckhafer, R.M.; Walter, C.J.; Finn, A.M.; and Thambidurai, P.M.: The MAFT Architecture for Distributed Fault Tolerance. *IEEE Transactions on Computers*, vol. 37, no. 4, Apr. 1988, pp. 398–405.

[27] Kopetz, Hermann; et al.: Distributed Fault-Tolerant Real-Time Systems: The Mars Approach. *IEEE Micro*, vol. 9, no. 1, Feb. 1989, pp. 25–40.

[28] Wensley, John H.; et al.: SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *Proceedings of the IEEE*, vol. 66, no. 10, Oct. 1978, pp. 1240–1255.

[29] Pease, M.; Shostak, R.; and Lamport, L.: Reaching Agreement in the Presence of Faults. *Journal of the ACM*, vol. 27, no. 2, April 1980, pp. 228–234.

[30] Garman, John R.: The “Bug” Heard ’Round the World. *ACM Software Engineering Notes*, vol. 6, no. 5, Oct. 1981, pp. 3–10.

[31] Anonymous: Reprogramming Capability Proves Key to Extending Voyager 2’s Journey. *Aviation Week and Space Technology*, August 7, 1989, pp. 72.

[32] Butler, Ricky W.: *A Survey of Provably Correct Fault-Tolerant Clock Synchronization Techniques.* NASA TM-100553, February 1988.

[33] Kopetz, Herman; and Ochsenreiter, Wilhelm: Clock Synchronization in Distributed Real-Time Systems. *IEEE Transactions on Computers*, vol. C-36, no. 8, Aug. 1987, pp. 933–940.

[34] Lamport, L.; and Melliar-Smith, P.M.: Synchronizing Clocks in the Presence of Faults. *Journal of the ACM*, vol. 32, no. 1, January 1985, pp. 52–78.

[35] Ramanathan, Parameswaran; Shin, Kang G.; and Butler, Ricky W.: Fault-Tolerant Clock Synchronization in Distributed Systems. *IEEE Computer*, vol. 23, no. 10, Oct. 1990, pp. 33–42.

[36] Schneider, Fred B.: *Understanding Protocols for Byzantine Clock Synchronization.* Department of Computer Science, Cornell University, Technical Report 87-859, Ithaca, NY, August 1987.

[37] Srikanth, T.K.; and Toueg, Sam: Optimal Clock Synchronization. *Journal of the ACM*, vol. 34, no. 3, July 1987, pp. 626–645.

[38] Rushby, John; and von Henke, Friedrich: *Formal Verification of a Fault Tolerant Clock Synchronization Algorithm.* NASA CR-4239, June 1989.

[39] Infis, A.H.; and Moore, W.R.: Economic Approach to Fault-Tolerant Synchronization. *IEE Proceedings, Part E*, vol. 135, no. 2, Mar. 1988, pp. 82–86.

[40] Cristian, Flaviu: *Probabilistic Clock Synchronization.* IBM Almaden Research Center, Technical Report RJ 6432, San Jose, CA, September 1988.

[41] Lamport, Leslie; Shostak, Robert; and Pease, Marshall: The Byzantine Generals Problem. *ACM TOPLAS*, vol. 4, no. 3, July 1982, pp. 382–401.

[42] Bevier, William R.; and Young, William D.: *Machine Checked Proofs of the Design and Implementation of a Fault-Tolerant Circuit.* NASA CR-182099, Nov. 1990.

[43] Bevier, W.R.; and Young, W.D.: The Design and Proof of Correctness of a Fault-Tolerant Circuit. In *2nd. International Working Conference on Dependable Computing for Critical Applications*, Tucson, AZ, Feb. 1991, IFIP WG. 10.4, pp. 107–114.

[44] Deckert, James C.; Desai, Mukund N.; Deyst, John J.; and Willsky, Alan S.: F-8 DFBW Sensor Failure Identification Using Analytic Redundancy. *IEEE Transactions on Automatic Control*, vol. AC-22, no. 5, Oct. 1977, pp. 795–803.

[45] DeLaat, John C.; and Merrill, Walter C.: A Real Time Microcomputer Implementation of Sensor Failure Detection for Turbofan Engines. *IEEE Control Systems Magazine*, vol. 10, no. 4, June 1990, pp. 29–37.

[46] Isermann, Rolf: Process Fault Detection Based on Modeling and Estimation Methods—A Survey. *Automatica*, vol. 20, no. 4, July 1984, pp. 387–404.

[47] Merrill, Walter C.; DeLaat, John C.; and Abdelwahab, Mahmood: Turbofan Engine Demonstration of Sensor Failure Detection. *AIAA Journal of Guidance, Control, and Dynamics*, vol. 14, no. 2, March-April 1991, pp. 337–349.

[48] Ray, Asok; and Luck, Rogelio: An Introduction to Sensor Signal Validation in Redundant Measurement Systems. *IEEE Control Systems Magazine*, vol. 11, no. 2, Feb. 1991, pp. 44–49.

[49] Willsky, Alan S.: A Survey of Methods for Failure Detection in Dynamic Systems. *Automatica*, vol. 12, no. 6, Nov. 1976, pp. 601–611.

[50] Chandraskeran, B.; and Punch III, W.F.: Data Validation During Diagnosis, A Step Beyond Traditional Sensor Validation. In *Proceedings, AAAI 87 (Volume 2)*, Seattle, WA, July 1987, pp. 778–782.

[51] Scarl, Ethan A.; Jamieson, John R.; and Delaune, Carl I.: Diagnosis and Sensor Validation through Knowledge of Structure and Function. *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-17, no. 3, May/June 1987, pp. 360–368.

[52] Marzullo, Keith: *Tolerating Failures of Continuous-Valued Sensors.* Department of Computer Science, Cornell University, Technical Report 90-1156, Ithaca, NY, Sept. 1990.

[53] Moser, L.; Melliar-Smith, P.M.; and Schwartz, R.: *Design Verification of SIFT.* NASA CR-4097, September 1987.

[54] Williams, John: Built to Last. *Astronomy Magazine*, vol. 18, no. 12, Dec. 1990, pp. 36–41.

[55] Kopetz, H.; Kantz, H.; Grünsteidl, G.; Puschner, P.; and Reisinger, J.: Tolerating Transient Faults in MARS. In *Digest of Papers, FTCS 20*, Newcastle upon Tyne, UK, June 1990, IEEE Computer Society, pp. 466–473.

[56] Shankar, Natarajan: *Mechanical Verification of a Schematic Protocol for Byzantine Fault-Tolerant Clock Synchronization.* Forthcoming NASA Contractor Report, 1991.

[57] Boyer, Robert S.; and Moore, J Strother: *MJRTY—A Fast Majority Vote Algorithm.* Institute for Computing Science, University of Texas, Technical Report 32, Austin TX, Feb. 1981.

[58] Meyer, Fred J.; and Pradhan, Dhiraj K.: Consensus with Dual Failure Modes. *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 2, Apr. 1991, pp. 214–222.

[59] Thambidurai, Philip; and Park, You-Keun: Interactive Consistency with Multiple Failure Modes. In *Proc. 7th Symposium on Reliable Distributed System*, Columbus, OH, Oct. 1988, IEEE Computer Society, pp. 93–100.

[60] Lamport, L.: The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks*, vol. 2, 1978, pp. 95–114.

[61] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, vol. 21, no. 7, July 1978, pp. 558–565.

[62] Lamport, L.: Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM TOPLAS*, vol. 6, no. 2, April 1984, pp. 254–280.

[63] Schneider, Fred B.: Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, vol. 22, no. 4, Dec. 1990, pp. 299–319.

Appendix A

LaTeX-printed Specification Listings

The following specification listings were formatted and converted to mathematical notation automatically using the EHDM LaTeX-printer.

simple_machine: **Module**

Exporting all

Theory

n: **Var** nat

M: **Type is** nat

m: **Var** M

C, D: **Type**

a, c: **Var** C

cell_types: **Type** = (sensor_cell, actuator_cell, task_cell)

cell_type: function[$C \rightarrow$ cell_types]

C_S : **Type from** C **with** ($\lambda\, c$: cell_type(c) = sensor_cell)

C_A : **Type from** C **with** ($\lambda\, c$: cell_type(c) = actuator_cell)

C_T : **Type from** C **with** ($\lambda\, c$: cell_type(c) $\neq$ sensor_cell) start_cell: C_S

arb_task: C_T

arb_actuator: C_A

$(\star 1, \star 2) \in \overline{G}$: function$[C, C \rightarrow$ bool$]$

sensor_ax: **Axiom** $(\exists\, a : (a, c) \in \overline{G}) \Leftrightarrow \neg(c$ **in** $C_S)$

sched: function$[M \rightarrow C]$

when: function$[C \rightarrow M]$

Gbar_when: **Axiom** $(a, c) \in \overline{G} \supset$ when$(a) <$ when(c)

sched_when_ax: **Axiom** (sched$(m) = a) \Leftrightarrow (m =$ when$(a))$

dowhen_pos: **Axiom** when$(c) > 0$

p, q: **Var** M

unique_when: **Lemma** $p \neq q \supset$ sched$(p) \neq$ sched(q)

previous: function$[C \rightarrow C]$ == $(\lambda\, c :$ sched(pred(when$(c))))$

sched_when_lemma: **Lemma** $a =$ sched(when$(a))$

when_sched_lemma: **Lemma** $m =$ when(sched$(m))$

dowhen_previous: **Lemma** when(previous$(c)) =$ pred(when$(c))$

state: **Type is** function$[C \rightarrow D]$

initial_state: state

s, t: **Var** state

sensor_fn: **Type is** function$[M \rightarrow D]$

sensor: function$[C_S \rightarrow$ sensor_fn$]$

task_fn: **Type is** function[state $\rightarrow D]$

task: function$[C_T \rightarrow$ task_fn$]$

```
  dependency: Axiom
    $c$ in $C_T \wedge (\forall a : (a, c) \in \overline{G} \supset s(a) = t(a))$
      $\supset$ task$(c)(s)$ = task$(c)(t)$

  step: function[state, $C, M \rightarrow$ state] =
    $(\lambda s, c, m : s$
        with [$(c)$ :=
              if $c$ in $C_S$ then sensor$(c)(m)$ else task$(c)(s)$ end if])

  identity: function[$M \rightarrow$ nat] == $(\lambda m : m)$

  run: Recursive function[$M \rightarrow$ state] =
    $(\lambda m :$
        if $m = 0$ then initial_state else step(run$(m - 1)$, sched$(m), m)$ end if)
    by identity

  runto: function[$C \rightarrow$ state] == $(\lambda c :$ run(when$(c)$))
```

Proof

```
  sched_when_proof: Prove sched_when_lemma from
    sched_when_ax $\{m \leftarrow$ when$(a)\}$

  when_sched_proof: Prove when_sched_lemma from
    sched_when_ax $\{a \leftarrow$ sched$(m)\}$

  dowhen_prev_proof: Prove dowhen_previous from
    when_sched_lemma $\{m \leftarrow$ pred(when$(c)$)$\}$

  unique_when_proof: Prove unique_when from
    when_sched_lemma $\{m \leftarrow p\}$, when_sched_lemma $\{m \leftarrow q\}$
```

End simple_machine

simple_machine_tcc: **Module**

Using simple_machine

Exporting all withsimple_machine

Theory

m: **Var** naturalnumber

a: **Var** C

c: **Var** C

s: **Var** function$[C \rightarrow D]$

t: **Var** function$[C \rightarrow D]$

sensors_TCC1: **Formula** $(\exists\, c : \text{cell\_type}(c) = \text{sensor\_cell})$

actuators_TCC1: **Formula** $(\exists\, c : \text{cell\_type}(c) = \text{actuator\_cell})$

active_tasks_TCC1: **Formula** $(\exists\, c : \text{cell\_type}(c) \neq \text{sensor\_cell})$

dependency_TCC1: **Formula**
$(c \textbf{ in } C_T \wedge (\forall\, a : (a, c) \in \overline{G} \supset s(a) = t(a)))$
$\supset (\text{cell\_type}(c) \neq \text{sensor\_cell})$

step_TCC1: **Formula** $(c \textbf{ in } C_S) \supset (\text{cell\_type}(c) = \text{sensor\_cell})$

step_TCC2: **Formula** $(\neg(c \textbf{ in } C_S)) \supset (\text{cell\_type}(c) \neq \text{sensor\_cell})$

run_TCC1: **Formula** $(\neg(m = 0)) \supset (m - 1 \geq 0)$

run_TCC2: **Formula** $(\neg(m = 0)) \supset \text{identity}(m) > \text{identity}(m - 1)$

Proof

sensors_TCC1_PROOF: **Prove** sensors_TCC1

actuators_TCC1_PROOF: **Prove** actuators_TCC1

active_tasks_TCC1_PROOF: **Prove** active_tasks_TCC1

dependency_TCC1_PROOF: **Prove** dependency_TCC1

step_TCC1_PROOF: **Prove** step_TCC1

step_TCC2_PROOF: **Prove** step_TCC2

run_TCC1_PROOF: **Prove** run_TCC1

run_TCC2_PROOF: **Prove** run_TCC2

End simple_machine_tcc

simple_machine_tcc_proofs: **Module**

Proof

Using simple_machine_tcc

sensors_TCC1_PROOF: **Prove** sensors_TCC1 {$c \leftarrow$ start_cell}

active_tasks_TCC1_PROOF: **Prove** active_tasks_TCC1 {$c \leftarrow$ arb_task}
from distinct_cell_types

actuators_TCC1_PROOF: **Prove** actuators_TCC1 {$c \leftarrow$ arb_actuator}

End simple_machine_tcc_proofs

noetherian: **Module** [dom: **Type**, <: function[dom, dom → bool]]

Assuming

measure: **Var** function[dom → nat]

a, b: **Var** dom

well_founded: **Formula**
$(\exists \text{ measure} : a < b \supset \text{measure}(a) < \text{measure}(b))$

Theory

p, A, B: **Var** function[dom → bool]

d, d_1, d_2: **Var** dom

general_induction: **Axiom**
$(\forall\, d_1 : (\forall\, d_2 : d_2 < d_1 \supset p(d_2)) \supset p(d_1)) \supset (\forall\, d : p(d))$

d_3, d_4: **Var** dom

mod_induction: **Theorem**
$(\forall\, d_3, d_4 : d_4 < d_3 \supset A(d_3) \supset A(d_4))$
$\quad \wedge (\forall\, d_1 : (\forall\, d_2 : d_2 < d_1 \supset (A(d_1) \wedge B(d_2))) \supset B(d_1))$
$\supset (\forall\, d : A(d) \supset B(d))$

Proof

mod_proof: **Prove**
mod_induction $\{d_1 \leftarrow d_1@p1,\ d_3 \leftarrow d_1@p1,\ d_4 \leftarrow d_2\}$
from general_induction $\{p \leftarrow (\lambda\, d : A(d) \supset B(d))\}$

End noetherian

natinduction: **Module**

Theory

i, m, m_1, n: **Var** nat

p: **Var** function[nat $\rightarrow$ bool]

induction: **Theorem** $(p(0) \wedge (\forall\, i : p(i) \supset p(i+1))) \supset p(n)$

induction_m: **Theorem**
$p(m) \wedge (\forall\, i : i \geq m \wedge p(i) \supset p(i+1)) \supset (\forall\, n : n \geq m \supset p(n))$

limited_induction: **Theorem**
$(m \leq m_1 \supset p(m)) \wedge (\forall\, i : i \geq m \wedge i < m_1 \wedge p(i) \supset p(i+1))$
$\supset (\forall\, n : n \geq m \wedge n \leq m_1 \supset p(n))$

Proof

Using noetherian

prev: function[nat, nat $\rightarrow$ bool] == $(\lambda\, m, n : m + 1 = n)$

instance: **Module is** noetherian[nat, prev]

x: **Var** nat

identity: function[nat $\rightarrow$ nat] == $(\lambda\, n : n)$

discharge: **Prove** well_founded {measure $\leftarrow$ identity}

ind_proof: **Prove** induction $\{i \leftarrow \text{pred}(d_1@p1)\}$ **from**
general_induction $\{d \leftarrow n,\ d_2 \leftarrow i\}$

ind_m_proof: **Prove** induction_m $\{i \leftarrow i@p1 + m\}$ **from**
induction
$\{p \leftarrow (\lambda\, x : \text{p@c}(x + m)),$
$n \leftarrow$ **if** $n \geq m$ **then** $n - m$ **else** 0 **end if**}

limited_proof: **Prove** limited_induction $\{i \leftarrow i@p1\}$ **from**
induction_m $\{p \leftarrow (\lambda\, x : x \leq m_1 \supset \text{p@c}(x))\}$

End natinduction

natinduction_tcc: **Module**

Using natinduction

Exporting all withnatinduction

Theory

m: **Var** naturalnumber

n: **Var** naturalnumber

ind_m_proof_TCC1: **Formula**
$(m \geq 0) \wedge (n \geq 0) \supset (\textbf{ if } n \geq m \textbf{ then } n - m \textbf{ else } 0 \textbf{ end if} \geq 0)$

Proof

ind_m_proof_TCC1_PROOF: **Prove** ind_m_proof_TCC1

End natinduction_tcc

simple_props: **Module**

Using simple_machine, natinduction

Exporting withsimple_machine

Theory

a, c: **Var** C

stay_correct_simple: **Lemma**
$(a, c) \in \overline{G} \supset \text{runto}(\text{previous}(c))(a) = \text{runto}(a)(a)$

simple_sensor_step_lemma: **Lemma**
c **in** $C_S \supset \text{runto}(c)(c) = \text{sensor}(c)(\text{when}(c))$

simple_step_lemma: **Lemma**
$\neg(c$ **in** $C_S) \supset \text{runto}(c)(c) = \text{task}(c)(\text{run}(\text{pred}(\text{when}(c))))$

Proof

m: **Var** M

indstep: **Lemma** $\text{run}(m)(a) = \text{runto}(a)(a) \supset \text{run}(m+1)(a) = \text{runto}(a)(a)$

indstep_proof: **Prove** indstep **from**
run $\{m \leftarrow m + 1\}$,
step $\{s \leftarrow \text{run}(m),\ c \leftarrow \text{sched}(m+1),\ m \leftarrow m+1\}$,
unique_when $\{p \leftarrow \text{when}(a),\ q \leftarrow m + 1\}$,
sched_when_lemma

q: **Var** M

stay_simple_proof: **Prove** stay_correct_simple **from**
induction_m
$\{p \leftarrow (\lambda\, q : \text{run}(q)(a) = \text{runto}(a)(a)),$
$m \leftarrow \text{when}(a),$
$n \leftarrow \text{when}(\text{previous}(c))\},$
indstep $\{m \leftarrow i@p1\}$,
sched_when_lemma $\{a \leftarrow \text{previous}(c)\}$,
Gbar_when,
when_sched_lemma $\{m \leftarrow \text{pred}(\text{when}(c))\}$

```
  simple_sensor_step_proof: Prove simple_sensor_step_lemma from
    run {m ← when(c)},
    step {s ← run(pred(when(c))), m ← when(c), c ← c},
    sched_when_lemma {a ← c},
    dowhen_pos

  simple_step_lemma_proof: Prove simple_step_lemma from
    run {m ← when(c)},
    step {m ← when(c), s ← run(pred(when(c)))},
    sched_when_lemma {a ← c},
    dowhen_pos

End simple_props
```

simple_props_tcc: **Module**

Using simple_props

Exporting all withsimple_props

Theory

c: **Var** simple_machine.C

i: **Var** naturalnumber

simple_sensor_step_lemma_TCC1: **Formula**
(c **in** C_S) $\supset$ (cell_type(c) = sensor_cell)

simple_step_lemma_TCC1: **Formula**
($\neg$(c **in** C_S)) $\supset$ (cell_type(c) $\neq$ sensor_cell)

Proof

simple_sensor_step_lemma_TCC1_PROOF: **Prove**
simple_sensor_step_lemma_TCC1

simple_step_lemma_TCC1_PROOF: **Prove** simple_step_lemma_TCC1

End simple_props_tcc

sets: **Module** [T: **Type**]

Exporting all

Theory

set: **Type is** function[$T \to$ bool]

x, y, z: **Var** T

a, b: **Var** set

$\star 1 \cup \star 2$: function[set, set $\to$ set] ==
$(\,\lambda\, a, b : (\,\lambda\, x : a(x) \vee b(x)))$

$\star 1 \cap \star 2$: function[set, set $\to$ set] ==
$(\,\lambda\, a, b : (\,\lambda\, x : a(x) \wedge b(x)))$

$\star 1 \setminus \star 2$: function[set, set $\to$ set] ==
$(\,\lambda\, a, b : (\,\lambda\, x : a(x) \wedge \neg b(x)))$

add: function[T, set $\to$ set] == $(\,\lambda\, x, a : (\,\lambda\, y : x = y \vee a(y)))$

$\{\star 1\}$: function[$T \to$ set] == $(\,\lambda\, x : (\,\lambda\, y : y = x))$

$\star 1 \subseteq \star 2$: function[set, set $\to$ bool] =
$(\,\lambda\, a, b : (\,\forall\, z : a(z) \supset b(z)))$

$\star 1 \in \star 2$: function[T, set $\to$ bool] == $(\,\lambda\, x, b : b(x))$

empty: function[set $\to$ bool] = $(\,\lambda\, a : (\,\forall\, x : \neg a(x)))$

$\emptyset$: set == $(\,\lambda\, x :$ false$)$

fullset: set == $(\,\lambda\, x :$ true$)$

extensionality: **Axiom** $(\,\forall\, x : x \in a = x \in b) \supset (a = b)$

End sets

cardinality: **Module** [T: **Type**]

Using sets[T]

Exporting all

Assuming

x, y, z: **Var** T

N: **Var** nat

f: **Var** function[$T \to$ nat]

finite: **Formula**
$(\exists N, f : (\forall x, y : f(x) \leq N \wedge (f(x) = f(y) \supset x = y)))$

Theory

a, b, c: **Var** set

$|\star 1|$: function[set $\to$ nat]

card_ax: **Axiom** $|a \cup b| + |a \cap b| = |a| + |b|$

card_subset: **Axiom** $a \subseteq b \supset |a| \leq |b|$

card_empty: **Axiom** $|a| = 0 \Leftrightarrow \text{empty}(a)$

empty_prop: **Lemma** $|a| > 0 \supset (\exists x : x \in a)$

card_prop: **Lemma**
$a \subseteq c \wedge b \subseteq c \wedge 2 * |a| > |c| \wedge 2 * |b| > |c| \supset |a \cap b| > 0$

Proof

empty_prop_proof: **Prove** empty_prop $\{x \leftarrow x@p2\}$ **from**
card_empty, empty

subset_union: **Sublemma** $a \subseteq c \wedge b \subseteq c \supset a \cup b \subseteq c$

subset_union_proof: **Prove** subset_union **from**
$\star 1 \subseteq \star 2 \ \{z \leftarrow z@p3,\ b \leftarrow c\}$,
$\star 1 \subseteq \star 2 \ \{z \leftarrow z@p3,\ a \leftarrow b,\ b \leftarrow c\}$,
$\star 1 \subseteq \star 2 \ \{a \leftarrow a \cup b,\ b \leftarrow c\}$

m, n, p: **Var** nat

twice_prop: **Sublemma** $2 * m > p \wedge 2 * n > p \supset m + n > p$

twice_proof: **Prove** twice_prop

card_proof: **Prove** card_prop **from**
twice_prop $\{m \leftarrow |a|,\ n \leftarrow |b|,\ p \leftarrow |c|\}$,
card_ax,
subset_union,
card_subset $\{a \leftarrow a \cup b,\ b \leftarrow c\}$

End cardinality

orderedsets: **Module** [T: **Type**, $\leq$: function[T, T → bool]]

Using sets[T]

Exporting min **with**sets[T]

Assuming

  x, y, z: **Var** T

  reflexive: **Formula** $x \leq x$

  transitive: **Formula** $x \leq y \wedge y \leq z \supset x \leq z$

  antisymmetry: **Formula** $x \leq y \wedge y \leq x \supset x = y$

  dichotomy: **Formula** $x \leq y \vee y \leq x$

Theory

  a: **Var** set

  min: function[set → T]

  min_ax: **Axiom** min(a) $\in a \wedge (\forall x : x \in a \supset$ min(a) $\leq x)$

End orderedsets

repl_machine: **Module**

Using simple_machine, sets, cardinality

Exporting all withsimple_machine

Theory

n: **Var** nat

m: **Var** M

c: **Var** C

voted: **Type from** C

voted_ax: **Axiom**
$(c \textbf{ in } C_A \supset c \textbf{ in } \text{voted}) \wedge (c \textbf{ in } \text{voted} \supset \neg(c \textbf{ in } C_S))$

r: nat

R: **Type from** nat **with** $(\lambda\, n : n \leq r)$

i: **Var** R

F: function$[R \rightarrow$ function$[M \rightarrow$ bool$]]$

rstate: **Type is** function$[R \rightarrow$ state$]$

σ, τ: **Var** rstate

maj: function[rstate, $C \rightarrow D$]

A: **Var** set$[R]$

x: **Var** D

maj_ax: **Axiom**
$(\exists\, A : 2 * |A| > |\text{fullset}[R]| \wedge (\forall\, i : i \in A \supset \sigma(i)(c) = x))$
$\supset \text{maj}(\sigma, c) = x$

vote: function[rstate, $C, M \rightarrow$ rstate]

```
vote_ax: Axiom
  $\neg(F(i)(m))$
    $\supset$ vote($\sigma, c, m$)
      = if $c$ in voted
         then $\sigma$
         with [$(i)(c)$ := maj($\sigma, c$)]
         else $\sigma$
         end if

sstep: function[rstate, $C, M \rightarrow$ rstate]

sstep_ax: Axiom $\neg(F(i)(m)) \supset$ sstep($\sigma, c, m$)($i$) = step($\sigma(i), c, m$)

rstep: function[rstate, $C, M \rightarrow$ rstate] ==
  ( $\lambda\, \sigma, c, m$ : vote(sstep($\sigma, c, m$), $c, m$))

rrun: Recursive function[$M \rightarrow$ rstate] =
  ( $\lambda\, m$ :
       if $m = 0$
        then ( $\lambda\, i$ : initial_state)
        else rstep(rrun($m - 1$), sched($m$), $m$)
        end if)
  by identity

rrunto: function[$C \rightarrow$ rstate] == ( $\lambda\, c$ : rrun(when($c$)))
```

Proof

```
disharge_finite: Prove
  finite[R] {$f \leftarrow$ ( $\lambda\, i \rightarrow$ nat : $i$), $N \leftarrow r$} from
  R_invariant {R_var $\leftarrow x$@$c$}
```

End repl_machine

repl_machine_tcc: **Module**

Using repl_machine

Exporting all withrepl_machine

Theory

n: **Var** naturalnumber

m: **Var** naturalnumber

x: **Var** R

R_TCC1: **Formula** $(\exists\, n : n \leq r)$

rrun_TCC1: **Formula** $(\neg(m = 0)) \supset (m - 1 \geq 0)$

rrun_TCC2: **Formula** $(\neg(m = 0)) \supset \text{identity}(m) > \text{identity}(m - 1)$

Proof

R_TCC1_PROOF: **Prove** R_TCC1

rrun_TCC1_PROOF: **Prove** rrun_TCC1

rrun_TCC2_PROOF: **Prove** rrun_TCC2

End repl_machine_tcc

repl_machine_tcc_proofs: **Module**

Proof

Using repl_machine_tcc

R_TCC1_PROOF: **Prove** R_TCC1 $\{n \leftarrow r\}$

End repl_machine_tcc_proofs

supports: **Module**

Using repl_machine, orderedsets$[M$, naturalnumbers. $\leq]$, sets$[C]$

Exporting support, committed_to
withrepl_machine, orderedsets$[M$, naturalnumbers. $\leq]$, sets$[C]$

Theory

a, b, c: **Var** C

foundation: **Recursive** function$[C \rightarrow \text{set}[C]] =$
$(\lambda\, c :$
$(\lambda\, a :$
$c = a$
$\vee\, (\neg(c$ **in** voted $\vee\, c$ **in** $C_S)$
$\wedge\, (\exists\, b : (b, c) \in \overline{G} \wedge a \in \text{foundation}(b)))))$
by when

backup: function$[C \rightarrow \text{set}[C]] =$
$(\lambda\, c : (\lambda\, a : (\exists\, b : (b, c) \in \overline{G} \wedge a \in \text{foundation}(b))))$

support: function$[C \rightarrow \text{set}[C]] =$
$(\lambda\, c : (\lambda\, a : a \in \text{foundation}(c) \vee (c$ **in** voted $\wedge\, a \in \text{backup}(c))))$

Gbar_support: **Lemma** $(a, c) \in \overline{G} \supset a \in \text{support}(c)$

in_own_support: **Lemma** $c \in \text{support}(c)$

subset_support: **Lemma**
$\neg(a$ **in** voted$)\wedge (a, c) \in \overline{G} \supset \text{support}(a) \subseteq \text{support}(c)$

S, T: **Var** set$[C]$

i: **Var** R

t, m: **Var** M

critical_times: function$[C \rightarrow \text{set}[M]] ==$
$(\lambda\, c : (\lambda\, t : \text{sched}(t) \in \text{support}(c)))$

committed_to: function$[C \rightarrow M] == (\lambda\, c : \text{min}(\text{critical_times}(c)))$

commit_when_lemma: **Lemma** committed_to$(c) \leq$ when(c)

commit_support_lemma: **Lemma**
$a \in$ support$(c) \supset$ committed_to$(c) \leq$ when(a)

commit_Gbar_lemma: **Lemma**
$(a, c) \in \overline{G} \wedge \neg(a$ **in** voted$) \supset$ committed_to$(c) \leq$ committed_to(a)

Proof

discharge_reflexive: **Prove** reflexive

discharge_transitive: **Prove** transitive

discharge_antisymmetry: **Prove** antisymmetry

discharge_dichotomy: **Prove** dichotomy

support_backup: **Sublemma** $a \in$ support$(c) = (c = a \vee a \in$ backup$(c))$

support_backup_proof: **Prove** support_backup **from**
support,
backup $\{b \leftarrow b@p3\}$,
foundation $\{b \leftarrow b@p2\}$,
sensor_ax $\{a \leftarrow b@P2\}$

Gbar_support_prf: **Prove** Gbar_support **from**
support_backup, backup $\{b \leftarrow a\}$, foundation $\{c \leftarrow a\}$

in_own_support_proof: **Prove** in_own_support **from**
support_backup $\{a \leftarrow c\}$

found_support: **Sublemma** $\neg(c$ **in** voted$) \supset$ foundation$(c) =$ support(c)

found_support_proof: **Prove** found_support **from**
support $\{a \leftarrow x@p2\}$,
extensionality[C] $\{a \leftarrow$ foundation(c), $b \leftarrow$ support$(c)\}$

found_sub_support: **Sublemma** $(b, c) \in \overline{G} \supset$ foundation$(b) \subseteq$ support(c)

found_sub_support_proof: **Prove** found_sub_support **from**
$\star 1 \subseteq \star 2$ [C] $\{a \leftarrow$ foundation(b), $b \leftarrow$ support$(c)\}$,
support_backup $\{a \leftarrow z@p1\}$,
backup $\{b \leftarrow b@C,\ a \leftarrow z@P1\}$

subset_support_proof: **Prove** subset_support **from**
found_sub_support $\{b \leftarrow a\}$, found_support $\{c \leftarrow a\}$

```
  committed_lemma: Sublemma
    committed_to(c) ∈ critical_times(c)
      ∧ (∀t : t ∈ critical_times(c) ⊃ t ≥ committed_to(c))

  committed_proof: Prove committed_lemma from
    min_ax {a ← critical_times(c), x ← t}

  commit_when_proof: Prove commit_when_lemma from
    in_own_support,
    committed_lemma {t ← when(c)},
    sched_when_lemma {a ← c}

  commit_support_proof: Prove commit_support_lemma from
    committed_lemma {t ← when(a)}, sched_when_lemma

  commit_Gbar_lemma_proof: Prove commit_Gbar_lemma from
    subset_support,
    ⋆1 ⊆ ⋆2 [C]
      {a ← support(a),
       b ← support(c),
       z ← sched(committed_to(a))},
    committed_lemma {t ← committed_to(a)},
    committed_lemma {c ← a}

End supports
```

supports_tcc: **Module**

Using supports

Exporting all withsupports

Theory

a: **Var** simple_machine.C

c: **Var** simple_machine.C

z: **Var** simple_machine.C

x: **Var** simple_machine.C

b: **Var** simple_machine.C

foundation_TCC1: **Formula**
$((b, c) \in \overline{G}) \wedge (\neg(c \textbf{ in } \text{voted} \vee c \textbf{ in } C_S)) \wedge (\neg(c = a))$
$\supset \text{when}(c) > \text{when}(b)$

Proof

foundation_TCC1_PROOF: **Prove** foundation_TCC1

End supports_tcc

supports_tcc_proofs: **Module**

Proof

Using supports_tcc

  foundation_TCC1_PROOF: **Prove** foundation_TCC1 **from**
    Gbar_when $\{a \leftarrow b\}$

End supports_tcc_proofs

correctness: **Module**

Using supports, sets[R], cardinality[R]

Exporting all with supports, sets[R]

Theory

i, j: **Var** R

a, c: **Var** C

m: **Var** M

OK: function[$R \rightarrow$ set[C]] =
($\lambda\, i$:
($\lambda\, c$:
($\forall\, m$: committed_to(c) $\leq m \wedge m \leq$ when(c) $\supset \neg F(i)(m)$))))

working: function[$C \rightarrow$ set[R]] == ($\lambda\, c$: ($\lambda\, i$: OK(i)(c)))

MOK: function[$C \rightarrow$ bool] = ($\lambda\, c$: 2 $*$ |working(c)| $>$ |fullset[R]|)

safe: **Recursive** function[$C \rightarrow$ bool] =
($\lambda\, c$: MOK(c) $\wedge$ ($\forall\, a$: $(a, c) \in \overline{G} \supset$ safe(a))) **by** when

correct: function[$C \rightarrow$ bool] =
($\lambda\, c$: ($\forall\, j$: OK(j)(c) $\supset$ rrunto(c)(j)(c) = runto(c)(c)))

the_result: **Theorem** safe(c) $\supset$ correct(c)

End correctness

correctness_tcc: **Module**

Using correctness

Exporting all withcorrectness

Theory

a: **Var** simple_machine.C

c: **Var** simple_machine.C

safe_TCC1: **Formula** $((a, c) \in \overline{G}) \wedge (\mathrm{MOK}(c)) \supset \mathrm{when}(c) > \mathrm{when}(a)$

Proof

safe_TCC1_PROOF: **Prove** safe_TCC1

End correctness_tcc

correctness_tcc_proofs: **Module**

Proof

Using correctness_tcc

safe_TCC1_PROOF: **Prove** safe_TCC1 **from** Gbar_when

End correctness_tcc_proofs

connect: **Module**

Using correctness, natinduction, simple_props

Exporting all

Theory

a, c: **Var** C

j: **Var** R

a_correct_at_c: function$[C, C \rightarrow$ bool$]$ =
$(\lambda\, a, c :$
$(\forall\, j :$
$\mathrm{OK}(j)(c) \supset \mathrm{rrunto}(\mathrm{previous}(c))(j)(a) = \mathrm{runto}(\mathrm{previous}(c))(a)))$

stay_correct: **Lemma**
$(\forall\, a : (a, c) \in \overline{G} \supset \mathrm{safe}(c) \wedge \mathrm{correct}(a))$
$\supset (\forall\, a : (a, c) \in \overline{G} \supset \mathrm{a\_correct\_at\_c}(a, c))$

Proof

i: **Var** R

m: **Var** M

r_indstep: **Lemma**
$\mathrm{OK}(j)(c)$
$\wedge\, (a, c) \in \overline{G}$
$\wedge\, \mathrm{when}(a) \leq m$
$\wedge\, m < \mathrm{when}(c) \wedge \mathrm{rrun}(m)(j)(a) = \mathrm{rrunto}(a)(j)(a)$
$\supset \mathrm{rrun}(m + 1)(j)(a) = \mathrm{rrunto}(a)(j)(a)$

```
r_indstep_proof: Prove r_indstep from
  rrun {m ← m + 1},
  vote_ax
    {σ ← sstep(rrun(m), sched(m + 1), m + 1),
     c ← sched(m + 1),
     m ← m + 1,
     i ← j},
  sstep_ax
    {σ ← rrun(m),
     c ← sched(m + 1),
     m ← m + 1,
     i ← j},
  step {s ← rrun(m)(j), c ← sched(m + 1), m ← m + 1},
  unique_when {p ← when(a), q ← m + 1},
  sched_when_lemma,
  OK {i ← j, m ← m + 1},
  commit_support_lemma,
  Gbar_support


q: Var M


stay_correct_repl: Lemma
  (a, c) ∈ Ḡ ∧ OK(j)(c) ⊃ rrunto(previous(c))(j)(a) = rrunto(a)(j)(a)


stay_correct_repl_proof: Prove stay_correct_repl from
  limited_induction
    {p ← (λ q : rrun(q)(j)(a) = rrunto(a)(j)(a)),
     m ← when(a),
     m₁ ← when(c),
     n ← when(previous(c))},
  r_indstep {m ← i@p1},
  sched_when_lemma {a ← previous(c)},
  Gbar_when,
  when_sched_lemma {m ← pred(when(c))},
  dowhen_pos


Gbar_OK: Lemma (a, c) ∈ Ḡ ∧ ¬(a in voted) ⊃ (OK(i)(c) ⊃ OK(i)(a))
```

Gbar_OK_proof: **Prove** Gbar_OK **from**
$\star 1 \subseteq \star 2$ [C] $\{a \leftarrow \text{support}(a),\ b \leftarrow \text{support}(c)\}$,
OK $\{m \leftarrow m@P3\}$,
OK $\{c \leftarrow a\}$,
Gbar_when,
commit_Gbar_lemma,
subset_support

notvoted_transfer_correct: **Lemma**
$(a, c) \in \overline{G} \wedge \text{safe}(c) \wedge \neg(a \textbf{ in } \text{voted}) \wedge \text{correct}(a)$
$\supset \text{OK}(j)(c) \supset \text{rrunto}(a)(j)(a) = \text{runto}(a)(a)$

notvoted_proof: **Prove** notvoted_transfer_correct **from**
Gbar_OK $\{i \leftarrow j\}$, correct $\{c \leftarrow a\}$

torch_carried: **Lemma**
$(a, c) \in \overline{G} \wedge \text{safe}(c) \supset (\exists j : \text{OK}(j)(a) \wedge \text{OK}(j)(c))$

torch_proof: **Prove** torch_carried $\{j \leftarrow x@p2\}$ **from**
card_prop[R]
$\{a \leftarrow \text{working}(c),$
$b \leftarrow \text{working}(a),$
$c \leftarrow \text{fullset}[R]\}$,
empty_prop[R] $\{a \leftarrow \text{working}(c) \cap \text{working}(a)\}$,
safe,
safe $\{c \leftarrow a\}$,
MOK,
MOK $\{c \leftarrow a\}$,
$\star 1 \subseteq \star 2$ [R] $\{a \leftarrow \text{working}(c),\ b \leftarrow \text{fullset}[R]\}$,
$\star 1 \subseteq \star 2$ [R] $\{a \leftarrow \text{working}(a),\ b \leftarrow \text{fullset}[R]\}$

σ: **Var** rstate

vote_appln: **Lemma**
$\neg(F(i)(\text{when}(a))) \wedge a \textbf{ in } \text{voted}$
$\supset \text{vote}(\sigma, a, \text{when}(a))(i)(a) = \text{maj}(\sigma, a)$

vote_appln_proof: **Prove** vote_appln **from**
vote_ax $\{c \leftarrow a,\ m \leftarrow \text{when}(a)\}$

safe_at_a: **Lemma** $\text{OK}(i)(c) \wedge (a, c) \in \overline{G} \supset \neg(F(i)(\text{when}(a)))$

safe_at_a_proof: **Prove** safe_at_a **from**
OK $\{m \leftarrow \text{when}(a)\}$, Gbar_when, Gbar_support, commit_support_lemma

OK_OK: **Lemma**
$\text{safe}(c) \wedge \text{OK}(i)(c) \wedge \text{OK}(j)(c) \wedge (a, c) \in \overline{G} \wedge a$ **in** voted
$\supset \text{rrunto}(a)(i)(a) = \text{rrunto}(a)(j)(a)$

OK_OK_proof: **Prove** OK_OK **from**
rrun $\{m \leftarrow \text{when}(a)\}$,
sched_when_lemma,
nat_invariant $\{\text{nat\_var} \leftarrow \text{when}(a)\}$,
vote_appln $\{\sigma \leftarrow \text{sstep}(\text{rrun}(\text{pred}(\text{when}(a))), a, \text{when}(a))\}$,
safe_at_a,
vote_appln
$\{i \leftarrow j, \sigma \leftarrow \text{sstep}(\text{rrun}(\text{pred}(\text{when}(a))), a, \text{when}(a))\}$,
safe_at_a $\{i \leftarrow j\}$

voted_transfer_correct: **Lemma**
$(a, c) \in \overline{G} \wedge \text{safe}(c) \wedge a$ **in** voted $\wedge$ correct(a)
$\supset \text{OK}(j)(c) \supset \text{rrunto}(a)(j)(a) = \text{runto}(a)(a)$

voted_proof: **Prove** voted_transfer_correct **from**
OK_OK $\{i \leftarrow j@p2\}$,
torch_carried,
correct $\{c \leftarrow a,\ j \leftarrow j@p2\}$

unvoted_transfer_correct: **Lemma**
$(a, c) \in \overline{G} \wedge \text{safe}(c) \wedge \text{correct}(a)$
$\supset \text{OK}(j)(c) \supset \text{rrunto}(a)(j)(a) = \text{runto}(a)(a)$

unvoted_proof: **Prove** unvoted_transfer_correct **from**
voted_transfer_correct, notvoted_transfer_correct

stay_correct_proof: **Prove** stay_correct **from**
stay_correct_simple,
stay_correct_repl $\{j \leftarrow j@p3\}$,
a_correct_at_c,
when_sched_lemma $\{m \leftarrow \text{pred}(\text{when}(c))\}$,
unvoted_transfer_correct $\{j \leftarrow j@p3\}$

End connect

sensor_step: **Module**

Using correctness, simple_props

Exporting **with**correctness, simple_props

Theory

a, c: **Var** C

sensor_inductive_step: **Lemma**
c **in** $C_S \wedge (\forall a : (a, c) \in \overline{G} \supset \text{safe}(c) \wedge \text{correct}(a)) \supset \text{correct}(c)$

Proof

j: **Var** R

sensor_step_lemma: **Lemma**
$\text{when}(c) > 0 \wedge \neg(c$ **in** voted$)$
$\supset \text{OK}(j)(c)$
$\supset \text{rrunto}(c)(j) = \text{step}(\text{rrun}(\text{pred}(\text{when}(c)))(j), c, \text{when}(c))$

sensor_step_proof: **Prove** sensor_step_lemma **from**
rrun $\{m \leftarrow \text{when}(c)\}$,
vote_ax
$\{i \leftarrow j,$
$m \leftarrow \text{when}(c),$
$\sigma \leftarrow \text{sstep}(\text{rrun}(\text{pred}(\text{when}(c))), c, \text{when}(c))\}$,
sstep_ax
$\{i \leftarrow j,$
$\sigma \leftarrow \text{rrun}(\text{pred}(\text{when}(c))),$
$m \leftarrow \text{when}(c)\}$,
sched_when_lemma $\{a \leftarrow c\}$,
OK $\{i \leftarrow j,\ m \leftarrow \text{when}(c)\}$,
commit_when_lemma

sensor_rrunto_lemma: **Lemma**
$\text{when}(c) > 0 \wedge c$ **in** C_S
$\supset \text{OK}(j)(c) \supset \text{rrunto}(c)(j)(c) = \text{sensor}(c)(\text{when}(c))$

```
sensor_rrunto_proof: Prove sensor_rrunto_lemma from
  sensor_step_lemma,
  step
    {$s \leftarrow$ rrun(pred(when($c$)))($j$),
     $m \leftarrow$ when($c$),
     $c \leftarrow c$},
  voted_ax

main_sensor_lemma: Lemma
  when($c$) $> 0 \wedge c$ in $C_S \supset$ OK($j$)($c$) $\supset$ rrunto($c$)($j$)($c$) = runto($c$)($c$)

main_sensor_proof: Prove main_sensor_lemma from
  simple_sensor_step_lemma, sensor_rrunto_lemma

sensor_ind_step_proof: Prove sensor_inductive_step from
  dowhen_pos, main_sensor_lemma {$j \leftarrow j$@$p3$}, correct, sensor_ax

End sensor_step
```

sensor_step_tcc: **Module**

Using sensor_step

Exporting all withsensor_step

Theory

c: **Var** simple_machine.C

j: **Var** repl_machine.R

sensor_rrunto_lemma_TCC1: **Formula**
$(\mathrm{OK}(j)(c)) \wedge (\mathrm{when}(c) > 0 \wedge c \ \mathbf{in}\ C_S)$
$\supset (\mathrm{cell\_type}(c) = \mathrm{sensor\_cell})$

Proof

sensor_rrunto_lemma_TCC1_PROOF: **Prove** sensor_rrunto_lemma_TCC1

End sensor_step_tcc

nonvoted_step: **Module**

Using correctness, connect

Exporting withcorrectness, connect

Theory

a, c: **Var** C

j: **Var** R

nonvoted_inductive_step: **Lemma**
$\neg(c$ **in** $C_S)$
$\wedge \neg(c$ **in** voted$) \wedge (\forall\, a : (a, c) \in \overline{G} \supset$ safe$(c) \wedge$ correct$(a))$
$\supset$ correct(c)

nonvoted_task_OK: **Lemma**
$\neg(c$ **in** $C_S) \wedge (\forall\, a : (a, c) \in \overline{G} \supset$ a_correct_at_c$(a, c))$
$\supset$ OK$(j)(c)$
$\supset$ task(c)(rrunto(previous$(c))(j))$ = task(c)(runto(previous$(c)))$

all_correct_at_c: function$[C \rightarrow$ bool$]$ =
$(\lambda\, c : (\forall\, a : (a, c) \in \overline{G} \supset$ a_correct_at_c$(a, c)))$

Proof

nonvoted_task_OK_proof: **Prove** nonvoted_task_OK $\{a \leftarrow a@p2\}$ **from**
a_correct_at_c $\{a \leftarrow a@p2\}$,
dependency
$\{s \leftarrow$ rrun(pred(when$(c)))(j)$,
$t \leftarrow$ run(pred(when$(c)))\}$,
dowhen_previous

nonvoted_rrunto_task: **Lemma**
$\neg(c$ **in** $C_S)$
$\wedge \neg(c$ **in** voted$) \wedge (\forall\, a : (a, c) \in \overline{G} \supset$ a_correct_at_c$(a, c))$
$\supset$ OK$(j)(c) \supset$ rrunto$(c)(j)(c)$ = task(c)(rrun(pred(when$(c)))(j))$

2

nonvoted_rrunto_task_proof: **Prove** nonvoted_rrunto_task **from**
rrun $\{m \leftarrow \text{when}(c)\}$,
vote_ax
$\{\sigma \leftarrow \text{sstep}(\text{rrun}(\text{pred}(\text{when}(c))), c, \text{when}(c)),$
$m \leftarrow \text{when}(c),$
$i \leftarrow j\}$,
sstep_ax
$\{i \leftarrow j,$
$\sigma \leftarrow \text{rrun}(\text{pred}(\text{when}(c))),$
$m \leftarrow \text{when}(c)\}$,
step $\{m \leftarrow \text{when}(c),\ s \leftarrow \text{rrun}(\text{pred}(\text{when}(c)))(j)\}$,
sched_when_lemma $\{a \leftarrow c\}$,
OK $\{i \leftarrow j,\ m \leftarrow \text{when}(c)\}$,
commit_when_lemma,
dowhen_pos

link: **Lemma**
$\neg(c \textbf{ in } C_S) \wedge \neg(c \textbf{ in } \text{voted}) \wedge (\forall a : (a, c) \in \overline{G} \supset \text{a\_correct\_at\_c}(a, c))$
$\supset \text{OK}(j)(c) \supset \text{rrunto}(c)(j)(c) = \text{runto}(c)(c)$

link_proof: **Prove** link $\{a \leftarrow a@p6\}$ **from**
nonvoted_rrunto_task,
simple_step_lemma,
nonvoted_task_OK,
dowhen_previous,
all_correct_at_c $\{a \leftarrow a@p3\}$,
all_correct_at_c $\{a \leftarrow a@p1\}$

main_non_voted_lemma: **Lemma**
$\neg(c \textbf{ in } C_S)$
$\wedge \neg(c \textbf{ in } \text{voted}) \wedge (\forall a : (a, c) \in \overline{G} \supset \text{safe}(c) \wedge \text{correct}(a))$
$\supset \text{OK}(j)(c) \supset \text{rrunto}(c)(j)(c) = \text{runto}(c)(c)$

main_nonvoted_proof: **Prove** main_non_voted_lemma $\{a \leftarrow a@p2\}$ **from**
link, stay_correct $\{a \leftarrow a@p1\}$

nonvoted_ind_proof: **Prove** nonvoted_inductive_step $\{a \leftarrow a@p1\}$ **from**
main_non_voted_lemma $\{j \leftarrow j@p2\}$, correct

End nonvoted_step

nonvoted_step_tcc: **Module**

Using nonvoted_step

Exporting all withnonvoted_step

Theory

c: **Var** simple_machine.C

j: **Var** repl_machine.R

a: **Var** simple_machine.C

nonvoted_task_OK_TCC1: **Formula**

$$\begin{aligned}
&(\mathrm{OK}(j)(c))\\
&\qquad \wedge (\neg(c \textbf{ in } C_S) \wedge (\forall\, a : (a, c) \in \overline{G} \supset \text{a\_correct\_at\_c}(a, c)))\\
&\quad \supset (\text{cell\_type}(c) \neq \text{sensor\_cell})
\end{aligned}$$

nonvoted_rrunto_task_TCC1: **Formula**

$$\begin{aligned}
&(\mathrm{OK}(j)(c))\\
&\qquad \wedge (\neg(c \textbf{ in } C_S)\\
&\qquad\qquad \wedge \neg(c \textbf{ in } \text{voted})\\
&\qquad\qquad\quad \wedge (\forall\, a : (a, c) \in \overline{G} \supset \text{a\_correct\_at\_c}(a, c)))\\
&\quad \supset (\text{cell\_type}(c) \neq \text{sensor\_cell})
\end{aligned}$$

Proof

nonvoted_task_OK_TCC1_PROOF: **Prove** nonvoted_task_OK_TCC1

nonvoted_rrunto_task_TCC1_PROOF: **Prove** nonvoted_rrunto_task_TCC1

End nonvoted_step_tcc

voted_step: **Module**

Using correctness, connect, nonvoted_step

Exporting induction_body **with**correctness, connect

Theory

a, c: **Var** C

voted_inductive_step: **Lemma**
c **in** voted $\wedge$ ($\forall\, a : (a, c) \in \overline{G} \supset$ safe(c) $\wedge$ correct(a))
$\supset$ correct(c)

induction_body: function[$C \rightarrow$ bool] =
($\lambda\, c$: ($\forall\, a : (a, c) \in \overline{G} \supset$ safe(c) $\wedge$ correct(a)))

Proof

i, j: **Var** R

σ: **Var** rstate

m: **Var** M

voted_step_lemma: **Lemma**
c **in** voted
$\supset$ OK(j)(c)
$\supset$ sstep(rrun(pred(when(c))), c, when(c))(j)(c)
= task(c)(rrun(pred(when(c)))(j))

voted_step_proof: **Prove** voted_step_lemma **from**
sstep_ax
{$i \leftarrow j$,
$\sigma \leftarrow$ rrun(pred(when(c))),
$m \leftarrow$ when(c)},
step {$m \leftarrow$ when(c), $s \leftarrow$ rrun(pred(when(c)))(j)},
OK {$i \leftarrow j$, $m \leftarrow$ when(c)},
commit_when_lemma,
voted_ax

```
sstep_task_lemma: Lemma
  $c$ in voted $\wedge$ ($\forall a : (a, c) \in \overline{G} \supset$ a_correct_at_c($a, c$))
    $\supset$ OK($j$)($c$)
      $\supset$ sstep(rrun(pred(when($c$))), $c$, when($c$))($j$)($c$)
        = task($c$)(run(pred(when($c$))))

sstep_task_proof: Prove sstep_task_lemma $\{a \leftarrow a@p2\}$ from
  voted_step_lemma, nonvoted_task_OK, dowhen_previous, voted_ax

$x$: Var $D$

maj_lemma: Lemma
  MOK($c$) $\wedge$ ($\forall i$ : OK($i$)($c$) $\supset \sigma(i)(c) = x$) $\supset$ maj($\sigma, c$) = $x$

maj_proof: Prove maj_lemma $\{i \leftarrow i@p1\}$ from
  maj_ax $\{A \leftarrow$ working($c$)$\}$, MOK

vote_lemma: Lemma
  OK($j$)($c$)
      $\wedge$ MOK($c$)
        $\wedge$ $c$ in voted
          $\wedge$ committed_to($c$) $\leq m$
            $\wedge$ $m \leq$ when($c$)
              $\wedge$ ($\forall i$ : OK($i$)($c$) $\supset$ sstep($\sigma, c, m$)($i$)($c$) = $x$)
    $\supset$ rstep($\sigma, c, m$)($j$)($c$) = $x$

vote_lemma_proof: Prove vote_lemma $\{i \leftarrow i@p2\}$ from
  vote_ax $\{i \leftarrow j,\ \sigma \leftarrow$ sstep($\sigma, c, m$)$\}$,
  maj_lemma $\{\sigma \leftarrow$ sstep($\sigma, c, m$)$\}$,
  OK $\{i \leftarrow j\}$

rstep_task: Lemma
  MOK($c$)
      $\wedge$ $c$ in voted
        $\wedge$ OK($j$)($c$) $\wedge$ ($\forall a : (a, c) \in \overline{G} \supset$ a_correct_at_c($a, c$))
    $\supset$ rstep(rrun(pred(when($c$))), $c$, when($c$))($j$)($c$)
      = task($c$)(run(pred(when($c$))))

active_task: function[$C \rightarrow C_T$] ==
  ($\lambda\, c \rightarrow C_T$ :  if $c$ in $C_S$ then arb_task else $c$ end if)
```

rstep_task_proof: **Prove** rstep_task $\{a \leftarrow a@p1\}$ **from**
sstep_task_lemma $\{j \leftarrow i@p2\}$,
vote_lemma
$\{x \leftarrow$ task(active_task(c))(run(pred(when(c)))),
$\sigma \leftarrow$ rrun(pred(when(c))),
$m \leftarrow$ when(c)$\}$,
commit_when_lemma,
voted_ax

rrunto_task: **Lemma**
MOK(c)
$\wedge$ c **in** voted
$\wedge$ OK(j)(c) $\wedge$ ($\forall\, a : (a, c) \in \overline{G} \supset$ a_correct_at_c(a, c))
$\supset$ rrunto(c)(j)(c) = task(c)(run(pred(when(c))))

rrunto_task_proof: **Prove** rrunto_task $\{a \leftarrow a@p1\}$ **from**
rstep_task,
rrun $\{m \leftarrow$ when(c)$\}$,
dowhen_pos,
sched_when_lemma $\{a \leftarrow c\}$

voted_link_lemma: **Lemma**
c **in** voted $\wedge$ MOK(c) $\wedge$ ($\forall\, a : (a, c) \in \overline{G} \supset$ a_correct_at_c(a, c))
$\supset$ OK(j)(c) $\supset$ rrunto(c)(j)(c) = runto(c)(c)

voted_link_proof: **Prove** voted_link_lemma $\{a \leftarrow a@p1\}$ **from**
rrunto_task, simple_step_lemma, voted_ax

main_voted_lemma: **Lemma**
c **in** voted $\wedge$ induction_body(c)
$\supset$ OK(j)(c) $\supset$ rrunto(c)(j)(c) = runto(c)(c)

sensors_not_voted: **Lemma** c **in** voted $\supset \neg$(c **in** C_S)

sensors_not_voted_proof: **Prove** sensors_not_voted **from** voted_ax

main_vote_proof: **Prove** main_voted_lemma **from**
voted_link_lemma,
safe,
stay_correct $\{a \leftarrow a@p1\}$,
sensor_ax,
sensors_not_voted,
induction_body $\{a \leftarrow a@p3\}$,
induction_body $\{a \leftarrow a@p4\}$

voted_ind_step_proof: **Prove** voted_inductive_step $\{a \leftarrow a@p3\}$ **from**
main_voted_lemma $\{j \leftarrow j@p2\}$, correct, induction_body

End voted_step

voted_step_tcc

voted_step_tcc: **Module**

Using voted_step

Exporting all withvoted_step

Theory

c: **Var** simple_machine.C

j: **Var** repl_machine.R

i: **Var** repl_machine.R

a: **Var** simple_machine.C

voted_step_lemma_TCC1: **Formula**
$(\mathrm{OK}(j)(c)) \wedge (c$ **in** voted$) \supset (\text{cell_type}(c) \neq \text{sensor_cell})$

sstep_task_lemma_TCC1: **Formula**
$(\mathrm{OK}(j)(c))$
$\wedge\ (c$ **in** voted $\wedge\ (\forall\, a : (a, c) \in \overline{G} \supset \text{a\_correct\_at\_c}(a, c)))$
$\supset (\text{cell\_type}(c) \neq \text{sensor\_cell})$

rstep_task_TCC1: **Formula**
$(\mathrm{MOK}(c) \wedge c$ **in** voted
$\wedge\ \mathrm{OK}(j)(c) \wedge (\forall\, a : (a, c) \in \overline{G} \supset \text{a\_correct\_at\_c}(a, c)))$
$\supset (\text{cell\_type}(c) \neq \text{sensor\_cell})$

active_task_TCC1: **Formula**
(cell_type(**if** c **in** C_S **then** arb_task **else** c **end if**) $\neq$ sensor_cell)

Proof

voted_step_lemma_TCC1_PROOF: **Prove** voted_step_lemma_TCC1

sstep_task_lemma_TCC1_PROOF: **Prove** sstep_task_lemma_TCC1

rstep_task_TCC1_PROOF: **Prove** rstep_task_TCC1

active_task_TCC1_PROOF: **Prove** active_task_TCC1

End voted_step_tcc

voted_step_tcc_proofs: **Module**

Proof

Using voted_step_tcc

voted_step_lemma_TCC1_PROOF: **Prove** voted_step_lemma_TCC1 **from** voted_ax

sstep_task_lemma_TCC1_PROOF: **Prove** sstep_task_lemma_TCC1 **from** voted_ax

rstep_task_TCC1_PROOF: **Prove** rstep_task_TCC1 **from** voted_ax

End voted_step_tcc_proofs

correctness_proof: **Module**

Using correctness, voted_step, nonvoted_step, sensor_step,
noetherian$[C, (\star 1, \star 2) \in \overline{G}]$

Exporting withcorrectness

Proof

a, c: **Var** C

discharge_well_founded: **Prove** well_founded {measure ← when} **from**
Gbar_when $\{c \leftarrow b\}$

inductive_step: **Lemma**
$(\forall\, a : (a, c) \in \overline{G} \supset \text{safe}(c) \wedge \text{correct}(a)) \supset \text{correct}(c)$

almost_final_proof: **Prove** inductive_step $\{a \leftarrow a@p7\}$ **from**
sensor_inductive_step,
voted_inductive_step,
nonvoted_inductive_step,
induction_body $\{a \leftarrow a@p1\}$,
induction_body $\{a \leftarrow a@p2\}$,
induction_body $\{a \leftarrow a@p3\}$,
induction_body

final_proof: **Prove** the_result **from**
mod_induction
$\{A \leftarrow$ safe,
$B \leftarrow$ correct,
$d \leftarrow c$,
$d_2 \leftarrow a@p3\}$,
safe $\{a \leftarrow d_4@p1,\ c \leftarrow d_3@p1\}$,
inductive_step $\{c \leftarrow d_1@p1\}$

End correctness_proof

outputs: **Module**

Using correctness

Exporting all withcorrectness

Theory

c: **Var** C

j: **Var** R

actuators_correct: **Corollary**
c **in** voted $\wedge$ safe(c) $\wedge \neg F(j)$(when(c))
$\supset$ rrunto(c)(j)(c) = runto(c)(c)

Proof

a: **Var** C

σ: **Var** rstate

i: **Var** R

m: **Var** M

vote_gives_maj: **Lemma**
$\neg F(i)(m) \wedge a$ **in** voted $\supset$ vote(σ, a, m)(i)(a) = maj(σ, a)

vote_gives_maj_proof: **Prove** vote_gives_maj **from** vote_ax $\{c \leftarrow a\}$

rrun_gets_maj: **Lemma**
$\neg F(i)$(when(a)) $\wedge$ a **in** voted
$\supset$ rrunto(a)(i)(a) = maj(sstep(rrun(pred(when(a))), a, when(a)), a)

rrun_gets_maj_proof: **Prove** rrun_gets_maj **from**
rrun $\{m \leftarrow$ when(a)$\}$,
vote_gives_maj
$\{\sigma \leftarrow$ sstep(rrun(pred(when(a))), a, when(a)),
$m \leftarrow$ when(a)$\}$,
dowhen_pos $\{c \leftarrow a\}$,
sched_when_lemma

working_agreement: **Lemma**
$\neg F(i)$(when(a)) $\wedge \neg F(j)$(when(a)) $\wedge$ a **in** voted
$\supset$ rrunto(a)(i)(a) = rrunto(a)(j)(a)

working_agreement_proof: **Prove** working_agreement **from**
rrun_gets_maj, rrun_gets_maj $\{i \leftarrow j\}$

safe_OK: **Lemma** safe(c) $\supset$ ($\exists j$: OK(j)(c))

safe_OK_proof: **Prove** safe_OK $\{j \leftarrow x@p4\}$ **from**
safe,
MOK,
nat_invariant {nat_var ← |fullset[R]|},
empty_prop[R] $\{a \leftarrow$ working(c)$\}$

actuators_correct_proof: **Prove** actuators_correct **from**
the_result $\{c \leftarrow c@c\}$,
correct $\{j \leftarrow i@p3,\ c \leftarrow c@c\}$,
working_agreement $\{a \leftarrow c@c,\ i \leftarrow j@p4\}$,
safe_OK,
OK $\{m \leftarrow$ when(c), $i \leftarrow i@p3\}$,
commit_when_lemma

End outputs

Appendix B

Cross-Reference Listing

This Appendix provides a cross-reference listing to the identifiers declared in the EHDM specification. It should assist in reading and navigating the EHDM specifications in Appendix A.

Identifier	Declaration	Module
a_correct_at_c	defined-fn	connect
active_task	literal-fn	voted_step
active_tasks	subtype-with	simple_machine
active_tasks_TCC1	formula	simple_machine_tcc
active_tasks_TCC1_PROOF	prove	simple_machine_tcc
active_tasks_TCC1_PROOF	prove	simple_machine_tcc_proofs
active_task_TCC1	formula	voted_step_tcc
active_task_TCC1_PROOF	prove	voted_step_tcc
actuators	subtype-with	simple_machine
actuators_correct	formula	outputs
actuators_correct_proof	prove	outputs
actuators_TCC1	formula	simple_machine_tcc
actuators_TCC1_PROOF	prove	simple_machine_tcc
actuators_TCC1_PROOF	prove	simple_machine_tcc_proofs
add	literal-fn	sets
all_correct_at_c	defined-fn	nonvoted_step
almost_final_proof	prove	correctness_proof
antisymmetry	formula	orderedsets
arb_actuator	const	simple_machine
arb_task	const	simple_machine
backup	defined-fn	supports
C	type	simple_machine
card	function	cardinality
card_ax	axiom	cardinality
card_empty	axiom	cardinality
cardinality	module	cardinality
card_proof	prove	cardinality
card_prop	formula	cardinality
card_subset	axiom	cardinality
cell_type	function	simple_machine
cell_types	type	simple_machine
commit_Gbar_lemma	formula	supports
commit_Gbar_lemma_proof	prove	supports
commit_support_lemma	formula	supports
commit_support_proof	prove	supports
committed_lemma	formula	supports

Table B.1: EHDM Identifers used in the Specification (continues)

Identifier	Declaration	Module
committed_proof	prove	supports
committed_to	literal-fn	supports
commit_when_lemma	formula	supports
commit_when_proof	prove	supports
connect	module	connect
correct	defined-fn	correctness
correctness	module	correctness
correctness_proof	module	correctness_proof
correctness_tcc	module	correctness_tcc
correctness_tcc_proofs	module	correctness_tcc_proofs
critical_times	literal-fn	supports
D	type	simple_machine
dependency	axiom	simple_machine
dependency_TCC1	formula	simple_machine_tcc
dependency_TCC1_PROOF	prove	simple_machine_tcc
dichotomy	formula	orderedsets
difference	literal-fn	sets
discharge	prove	natinduction
discharge_antisymmetry	prove	supports
discharge_dichotomy	prove	supports
discharge_reflexive	prove	supports
discharge_transitive	prove	supports
discharge_well_founded	prove	correctness_proof
disharge_finite	prove	repl_machine
dowhen	function	simple_machine
dowhen_pos	axiom	simple_machine
dowhen_previous	formula	simple_machine
dowhen_prev_proof	prove	simple_machine
empty	defined-fn	sets
empty_prop	formula	cardinality
empty_prop_proof	prove	cardinality
emptyset	literal-const	sets
extensionality	axiom	sets
F	function	repl_machine
final_proof	prove	correctness_proof
finite	formula	cardinality

Table B.1: EHDM Identifers used in the Specification (continues)

Identifier	Declaration	Module
foundation	recursive-fn	supports
foundation_TCC1	formula	supports_tcc
foundation_TCC1_PROOF	prove	supports_tcc
foundation_TCC1_PROOF	prove	supports_tcc_proofs
found_sub_support	formula	supports
found_sub_support_proof	prove	supports
found_support	formula	supports
found_support_proof	prove	supports
fullset	literal-const	sets
Gbar	function	simple_machine
Gbar_OK	formula	connect
Gbar_OK_proof	prove	connect
Gbar_support	formula	supports
Gbar_support_prf	prove	supports
Gbar_when	axiom	simple_machine
general_induction	axiom	noetherian
identity	literal-fn	natinduction
identity	literal-fn	simple_machine
ind_m_proof	prove	natinduction
ind_m_proof_TCC1	formula	natinduction_tcc
ind_m_proof_TCC1_PROOF	prove	natinduction_tcc
ind_proof	prove	natinduction
indstep	formula	simple_props
indstep_proof	prove	simple_props
induction	formula	natinduction
induction_body	defined-fn	voted_step
induction_m	formula	natinduction
inductive_step	formula	correctness_proof
in_own_support	formula	supports
in_own_support_proof	prove	supports
instance	module	natinduction
intersection	literal-fn	sets
limited_induction	formula	natinduction
limited_proof	prove	natinduction
link	formula	nonvoted_step
link_proof	prove	nonvoted_step

Table B.1: EHDM Identifers used in the Specification (continues)

Identifier	Declaration	Module
M	type	simple_machine
main_non_voted_lemma	formula	nonvoted_step
main_nonvoted_proof	prove	nonvoted_step
main_sensor_lemma	formula	sensor_step
main_sensor_proof	prove	sensor_step
main_voted_lemma	formula	voted_step
main_vote_proof	prove	voted_step
maj	function	repl_machine
maj_ax	axiom	repl_machine
maj_lemma	formula	voted_step
maj_proof	prove	voted_step
member	literal-fn	sets
min	function	orderedsets
min_ax	axiom	orderedsets
mod_induction	formula	noetherian
mod_proof	prove	noetherian
MOK	defined-fn	correctness
natinduction	module	natinduction
natinduction_tcc	module	natinduction_tcc
noetherian	module	noetherian
nonvoted_ind_proof	prove	nonvoted_step
nonvoted_inductive_step	formula	nonvoted_step
nonvoted_rrunto_task	formula	nonvoted_step
nonvoted_rrunto_task_proof	prove	nonvoted_step
nonvoted_rrunto_task_TCC1	formula	nonvoted_step_tcc
nonvoted_rrunto_task_TCC1_PROOF	prove	nonvoted_step_tcc
nonvoted_step	module	nonvoted_step
nonvoted_step_tcc	module	nonvoted_step_tcc
nonvoted_task_OK	formula	nonvoted_step
nonvoted_task_OK_proof	prove	nonvoted_step
nonvoted_task_OK_TCC1	formula	nonvoted_step_tcc
nonvoted_task_OK_TCC1_PROOF	prove	nonvoted_step_tcc
notvoted_proof	prove	connect
notvoted_transfer_correct	formula	connect
OK	defined-fn	correctness
OK_OK	formula	connect

Table B.1: EHDM Identifers used in the Specification (continues)

Identifier	Declaration	Module
OK_OK_proof	prove	connect
orderedsets	module	orderedsets
outputs	module	outputs
prev	literal-fn	natinduction
previous	literal-fn	simple_machine
r	const	repl_machine
R	subtype-with	repl_machine
reflexive	formula	orderedsets
repl_machine	module	repl_machine
repl_machine_tcc	module	repl_machine_tcc
repl_machine_tcc_proofs	module	repl_machine_tcc_proofs
r_indstep	formula	connect
r_indstep_proof	prove	connect
rrun	recursive-fn	repl_machine
rrun_gets_maj	formula	outputs
rrun_gets_maj_proof	prove	outputs
rrun_TCC1	formula	repl_machine_tcc
rrun_TCC1_PROOF	prove	repl_machine_tcc
rrun_TCC2	formula	repl_machine_tcc
rrun_TCC2_PROOF	prove	repl_machine_tcc
rrunto	literal-fn	repl_machine
rrunto_task	formula	voted_step
rrunto_task_proof	prove	voted_step
rstate	type	repl_machine
rstep	literal-fn	repl_machine
rstep_task	formula	voted_step
rstep_task_proof	prove	voted_step
rstep_task_TCC1	formula	voted_step_tcc
rstep_task_TCC1_PROOF	prove	voted_step_tcc
rstep_task_TCC1_PROOF	prove	voted_step_tcc_proofs
R_TCC1	formula	repl_machine_tcc
R_TCC1_PROOF	prove	repl_machine_tcc
R_TCC1_PROOF	prove	repl_machine_tcc_proofs
run	recursive-fn	simple_machine
run_TCC1	formula	simple_machine_tcc
run_TCC1_PROOF	prove	simple_machine_tcc

Table B.1: EHDM Identifers used in the Specification (continues)

Identifier	Declaration	Module
run_TCC2	formula	simple_machine_tcc
run_TCC2_PROOF	prove	simple_machine_tcc
runto	literal-fn	simple_machine
safe	recursive-fn	correctness
safe_at_a	formula	connect
safe_at_a_proof	prove	connect
safe_OK	formula	outputs
safe_OK_proof	prove	outputs
safe_TCC1	formula	correctness_tcc
safe_TCC1_PROOF	prove	correctness_tcc
safe_TCC1_PROOF	prove	correctness_tcc_proofs
sched	function	simple_machine
sched_when_ax	axiom	simple_machine
sched_when_lemma	formula	simple_machine
sched_when_proof	prove	simple_machine
sensor	function	simple_machine
sensor_ax	axiom	simple_machine
sensor_fn	type	simple_machine
sensor_ind_step_proof	prove	sensor_step
sensor_inductive_step	formula	sensor_step
sensor_rrunto_lemma	formula	sensor_step
sensor_rrunto_lemma_TCC1	formula	sensor_step_tcc
sensor_rrunto_lemma_TCC1_PROOF	prove	sensor_step_tcc
sensor_rrunto_proof	prove	sensor_step
sensors	subtype-with	simple_machine
sensors_not_voted	formula	voted_step
sensors_not_voted_proof	prove	voted_step
sensors_TCC1	formula	simple_machine_tcc
sensors_TCC1_PROOF	prove	simple_machine_tcc
sensors_TCC1_PROOF	prove	simple_machine_tcc_proofs
sensor_step	module	sensor_step
sensor_step_lemma	formula	sensor_step
sensor_step_proof	prove	sensor_step
sensor_step_tcc	module	sensor_step_tcc
set	type	sets
sets	module	sets

Table B.1: EHDM Identifers used in the Specification (continues)

Identifier	Declaration	Module
simple_machine	module	simple_machine
simple_machine_tcc	module	simple_machine_tcc
simple_machine_tcc_proofs	module	simple_machine_tcc_proofs
simple_props	module	simple_props
simple_props_tcc	module	simple_props_tcc
simple_sensor_step_lemma	formula	simple_props
simple_sensor_step_lemma_TCC1	formula	simple_props_tcc
simple_sensor_step_lemma_TCC1_PROOF	prove	simple_props_tcc
simple_sensor_step_proof	prove	simple_props
simple_step_lemma	formula	simple_props
simple_step_lemma_proof	prove	simple_props
simple_step_lemma_TCC1	formula	simple_props_tcc
simple_step_lemma_TCC1_PROOF	prove	simple_props_tcc
singleton	literal-fn	sets
sstep	function	repl_machine
sstep_ax	axiom	repl_machine
sstep_task_lemma	formula	voted_step
sstep_task_lemma_TCC1	formula	voted_step_tcc
sstep_task_lemma_TCC1_PROOF	prove	voted_step_tcc
sstep_task_lemma_TCC1_PROOF	prove	voted_step_tcc_proofs
sstep_task_proof	prove	voted_step
start_cell	const	simple_machine
state	type	simple_machine
stay_correct	formula	connect
stay_correct_proof	prove	connect
stay_correct_repl	formula	connect
stay_correct_repl_proof	prove	connect
stay_correct_simple	formula	simple_props
stay_simple_proof	prove	simple_props
step	defined-fn	simple_machine
step_TCC1	formula	simple_machine_tcc
step_TCC1_PROOF	prove	simple_machine_tcc
step_TCC2	formula	simple_machine_tcc
step_TCC2_PROOF	prove	simple_machine_tcc
subset	defined-fn	sets
subset_support	formula	supports

Table B.1: EHDM Identifers used in the Specification (continues)

Identifier	Declaration	Module
subset_support_proof	prove	supports
subset_union	formula	cardinality
subset_union_proof	prove	cardinality
support	defined-fn	supports
support_backup	formula	supports
support_backup_proof	prove	supports
supports	module	supports
supports_tcc	module	supports_tcc
supports_tcc_proofs	module	supports_tcc_proofs
task	function	simple_machine
task_fn	type	simple_machine
the_result	formula	correctness
torch_carried	formula	connect
torch_proof	prove	connect
transitive	formula	orderedsets
twice_proof	prove	cardinality
twice_prop	formula	cardinality
undef	const	simple_machine
union	literal-fn	sets
unique_when	formula	simple_machine
unique_when_proof	prove	simple_machine
unvoted_proof	prove	connect
unvoted_transfer_correct	formula	connect
vote	function	repl_machine
vote_appln	formula	connect
vote_appln_proof	prove	connect
vote_ax	axiom	repl_machine
voted	subtype	repl_machine
voted_ax	axiom	repl_machine
voted_ind_step_proof	prove	voted_step
voted_inductive_step	formula	voted_step
voted_link_lemma	formula	voted_step
voted_link_proof	prove	voted_step
voted_proof	prove	connect
voted_step	module	voted_step
voted_step_lemma	formula	voted_step

Table B.1: EHDM Identifers used in the Specification (continues)

Identifier	Declaration	Module
voted_step_lemma_TCC1	formula	voted_step_tcc
voted_step_lemma_TCC1_PROOF	prove	voted_step_tcc
voted_step_lemma_TCC1_PROOF	prove	voted_step_tcc_proofs
voted_step_proof	prove	voted_step
voted_step_tcc	module	voted_step_tcc
voted_step_tcc_proofs	module	voted_step_tcc_proofs
voted_transfer_correct	formula	connect
vote_gives_maj	formula	outputs
vote_gives_maj_proof	prove	outputs
vote_lemma	formula	voted_step
vote_lemma_proof	prove	voted_step
well_founded	formula	noetherian
when_sched_lemma	formula	simple_machine
when_sched_proof	prove	simple_machine
working	literal-fn	correctness
working_agreement	formula	outputs
working_agreement_proof	prove	outputs

Table B.1: EHDM Identifers used in the Specification

Appendix C

Results of Proof-Chain Analysis

The following pages reproduce the output from the EHDM proof-chain analyzer in "terse mode" applied to the formula actuators_correct in module outputs. The EHDM proof-chain analyzer examines the macroscopic structure of a verification—checking that all the premises used in a proof are either axioms, definitions, or formulas which are, themselves, the target of a successful proof elsewhere in the verification. If any formulas are used from a module having an assuming clause, then the proof-chain analyzer checks that those assumptions are discharged by successful proofs; similarly, if formulas are used from a module having a tcc module, then the proof-chain analyzer checks that all the tccs in that module are discharged by successful proofs. The proof-chain analyzer ignores unsuccessful proofs (such as automatically-generated tcc proofs) when a successful proof for the same formula can be found. The "terse mode" output reproduced here provides a commentary on only the "interesting" cases, namely proof obligations involving assuming clauses and tccs, and a summary. All the proofs listed in the summary were performed by the EHDM theorem prover in "checking mode."

```
Proof chain for formula actuators_correct in module outputs

Use of the formula
  correctness.the_result
requires the following TCCs to be proven
  correctness_tcc.safe_TCC1

Formula correctness_tcc.safe_TCC1 is a termination TCC for correctness.safe
Proof of
  correctness_tcc.safe_TCC1
must not use
  correctness.safe

Use of the formula
  simple_machine.Gbar_when
requires the following TCCs to be proven
  simple_machine_tcc.sensors_TCC1
  simple_machine_tcc.actuators_TCC1
  simple_machine_tcc.active_tasks_TCC1
  simple_machine_tcc.dependency_TCC1
  simple_machine_tcc.step_TCC1
  simple_machine_tcc.step_TCC2
  simple_machine_tcc.run_TCC1
  simple_machine_tcc.run_TCC2

Formula simple_machine_tcc.run_TCC2 is a termination TCC for simple_machine.run
Proof of
  simple_machine_tcc.run_TCC2
must not use
  simple_machine.run

Use of the formula
  noetherian[simple_machine.C, simple_machine.Gbar].mod_induction
requires the following assumptions to be discharged
  noetherian[simple_machine.C, simple_machine.Gbar].well_founded

Use of the formula
  sensor_step.sensor_inductive_step
requires the following TCCs to be proven
  sensor_step_tcc.sensor_rrunto_lemma_TCC1

Use of the formula
  simple_props.simple_sensor_step_lemma
requires the following TCCs to be proven
  simple_props_tcc.simple_sensor_step_lemma_TCC1
  simple_props_tcc.simple_step_lemma_TCC1
```

```
Use of the formula
  repl_machine.rrun
requires the following TCCs to be proven
  repl_machine_tcc.R_TCC1
  repl_machine_tcc.rrun_TCC1
  repl_machine_tcc.rrun_TCC2

Formula repl_machine_tcc.rrun_TCC2 is a termination TCC for repl_machine.rrun
Proof of
  repl_machine_tcc.rrun_TCC2
must not use
  repl_machine.rrun

Use of the formula
  supports.commit_when_lemma
requires the following TCCs to be proven
  supports_tcc.foundation_TCC1

Formula supports_tcc.foundation_TCC1 is a termination TCC for
supports.foundation
Proof of
  supports_tcc.foundation_TCC1
must not use
  supports.foundation

Use of the formula
  orderedsets[naturalnumber, <=].min_ax
requires the following assumptions to be discharged
  orderedsets[naturalnumber, <=].reflexive
  orderedsets[naturalnumber, <=].transitive
  orderedsets[naturalnumber, <=].antisymmetry
  orderedsets[naturalnumber, <=].dichotomy

Use of the formula
  voted_step.voted_inductive_step
requires the following TCCs to be proven
  voted_step_tcc.voted_step_lemma_TCC1
  voted_step_tcc.sstep_task_lemma_TCC1
  voted_step_tcc.rstep_task_TCC1
  voted_step_tcc.active_task_TCC1

Use of the formula
  nonvoted_step.nonvoted_task_OK
requires the following TCCs to be proven
  nonvoted_step_tcc.nonvoted_task_OK_TCC1
```

```
    nonvoted_step_tcc.nonvoted_rrunto_task_TCC1

  Use of the formula
    natinduction.induction_m
  requires the following TCCs to be proven
    natinduction_tcc.ind_m_proof_TCC1

  Use of the formula
    noetherian[naturalnumber, natinduction.prev].general_induction
  requires the following assumptions to be discharged
    noetherian[naturalnumber, natinduction.prev].well_founded

  Use of the formula
    cardinality[repl_machine.R].card_prop
  requires the following assumptions to be discharged
    cardinality[repl_machine.R].finite

    ================== SUMMARY ==================

  The proof chain is complete

  The axioms and assumptions at the base are:
    cardinality[EXPR].card_ax
    cardinality[EXPR].card_empty
    cardinality[EXPR].card_subset
    naturalnumbers.nat_invariant
    noetherian[EXPR, EXPR].general_induction
    orderedsets[EXPR, EXPR].min_ax
    repl_machine.R_invariant
    repl_machine.maj_ax
    repl_machine.sstep_ax
    repl_machine.vote_ax
    repl_machine.voted_ax
    sets[EXPR].extensionality
    simple_machine.Gbar_when
    simple_machine.dependency
    simple_machine.distinct_cell_types
    simple_machine.dowhen_pos
    simple_machine.sched_when_ax
    simple_machine.sensor_ax
Total: 18

The definitions are:
  connect.a_correct_at_c
  correctness.MOK
  correctness.OK
  correctness.correct
  correctness.safe
  nonvoted_step.all_correct_at_c
  repl_machine.rrun
  sets[EXPR].empty
  sets[EXPR].subset
  simple_machine.run
  simple_machine.step
  supports.backup
  supports.foundation
  supports.support
  voted_step.induction_body
Total: 15

The formulae used are:
  cardinality[EXPR].card_prop
  cardinality[EXPR].empty_prop
  cardinality[EXPR].subset_union
  cardinality[EXPR].twice_prop
  cardinality[repl_machine.R].finite
  connect.Gbar_OK
  connect.OK_OK
  connect.notvoted_transfer_correct
  connect.r_indstep
  connect.safe_at_a
  connect.stay_correct
  connect.stay_correct_repl
  connect.torch_carried
  connect.unvoted_transfer_correct
  connect.vote_appln
  connect.voted_transfer_correct
  correctness.the_result
  correctness_proof.inductive_step
  correctness_tcc.safe_TCC1
  natinduction.induction
  natinduction.induction_m
  natinduction.limited_induction
  natinduction_tcc.ind_m_proof_TCC1
  noetherian[EXPR, EXPR].mod_induction
  noetherian[naturalnumber, natinduction.prev].well_founded
  noetherian[simple_machine.C, simple_machine.Gbar].well_founded
  nonvoted_step.link
  nonvoted_step.main_non_voted_lemma
  nonvoted_step.nonvoted_inductive_step
  nonvoted_step.nonvoted_rrunto_task
  nonvoted_step.nonvoted_task_OK
```

```
nonvoted_step_tcc.nonvoted_rrunto_task_TCC1
nonvoted_step_tcc.nonvoted_task_OK_TCC1
orderedsets[naturalnumber, <=].antisymmetry
orderedsets[naturalnumber, <=].dichotomy
orderedsets[naturalnumber, <=].reflexive
orderedsets[naturalnumber, <=].transitive
outputs.actuators_correct
outputs.rrun_gets_maj
outputs.safe_OK
outputs.vote_gives_maj
outputs.working_agreement
repl_machine_tcc.R_TCC1
repl_machine_tcc.rrun_TCC1
repl_machine_tcc.rrun_TCC2
sensor_step.main_sensor_lemma
sensor_step.sensor_inductive_step
sensor_step.sensor_rrunto_lemma
sensor_step.sensor_step_lemma
sensor_step_tcc.sensor_rrunto_lemma_TCC1
simple_machine.dowhen_previous
simple_machine.sched_when_lemma
simple_machine.unique_when
simple_machine.when_sched_lemma
simple_machine_tcc.active_tasks_TCC1
simple_machine_tcc.actuators_TCC1
simple_machine_tcc.dependency_TCC1
simple_machine_tcc.run_TCC1
simple_machine_tcc.run_TCC2
simple_machine_tcc.sensors_TCC1
simple_machine_tcc.step_TCC1
simple_machine_tcc.step_TCC2
simple_props.indstep
simple_props.simple_sensor_step_lemma
simple_props.simple_step_lemma
simple_props.stay_correct_simple
simple_props_tcc.simple_sensor_step_lemma_TCC1
simple_props_tcc.simple_step_lemma_TCC1
supports.Gbar_support
supports.commit_Gbar_lemma
supports.commit_support_lemma
supports.commit_when_lemma
supports.committed_lemma
supports.found_sub_support
supports.found_support
supports.in_own_support
supports.subset_support
```

```
  supports.support_backup
  supports_tcc.foundation_TCC1
  voted_step.main_voted_lemma
  voted_step.maj_lemma
  voted_step.rrunto_task
  voted_step.rstep_task
  voted_step.sensors_not_voted
  voted_step.sstep_task_lemma
  voted_step.vote_lemma
  voted_step.voted_inductive_step
  voted_step.voted_link_lemma
  voted_step.voted_step_lemma
  voted_step_tcc.active_task_TCC1
  voted_step_tcc.rstep_task_TCC1
  voted_step_tcc.sstep_task_lemma_TCC1
  voted_step_tcc.voted_step_lemma_TCC1
Total: 93

The completed proofs are:
  cardinality[EXPR].card_proof
  cardinality[EXPR].empty_prop_proof
  cardinality[EXPR].subset_union_proof
  cardinality[EXPR].twice_proof
  connect.Gbar_OK_proof
  connect.OK_OK_proof
  connect.notvoted_proof
  connect.r_indstep_proof
  connect.safe_at_a_proof
  connect.stay_correct_proof
  connect.stay_correct_repl_proof
  connect.torch_proof
  connect.unvoted_proof
  connect.vote_appln_proof
  connect.voted_proof
  correctness_proof.almost_final_proof
  correctness_proof.discharge_well_founded
  correctness_proof.final_proof
  correctness_tcc_proofs.safe_TCC1_PROOF
  natinduction.discharge
  natinduction.ind_m_proof
  natinduction.ind_proof
  natinduction.limited_proof
  natinduction_tcc.ind_m_proof_TCC1_PROOF
  noetherian[EXPR, EXPR].mod_proof
  nonvoted_step.link_proof
  nonvoted_step.main_nonvoted_proof
nonvoted_step.nonvoted_ind_proof
nonvoted_step.nonvoted_rrunto_task_proof
nonvoted_step.nonvoted_task_OK_proof
nonvoted_step_tcc.nonvoted_rrunto_task_TCC1_PROOF
nonvoted_step_tcc.nonvoted_task_OK_TCC1_PROOF
outputs.actuators_correct_proof
outputs.rrun_gets_maj_proof
outputs.safe_OK_proof
outputs.vote_gives_maj_proof
outputs.working_agreement_proof
repl_machine.disharge_finite
repl_machine_tcc.rrun_TCC1_PROOF
repl_machine_tcc.rrun_TCC2_PROOF
repl_machine_tcc_proofs.R_TCC1_PROOF
sensor_step.main_sensor_proof
sensor_step.sensor_ind_step_proof
sensor_step.sensor_rrunto_proof
sensor_step.sensor_step_proof
sensor_step_tcc.sensor_rrunto_lemma_TCC1_PROOF
simple_machine.dowhen_prev_proof
simple_machine.sched_when_proof
simple_machine.unique_when_proof
simple_machine.when_sched_proof
simple_machine_tcc.dependency_TCC1_PROOF
simple_machine_tcc.run_TCC1_PROOF
simple_machine_tcc.run_TCC2_PROOF
simple_machine_tcc.step_TCC1_PROOF
simple_machine_tcc.step_TCC2_PROOF
simple_machine_tcc_proofs.active_tasks_TCC1_PROOF
simple_machine_tcc_proofs.actuators_TCC1_PROOF
simple_machine_tcc_proofs.sensors_TCC1_PROOF
simple_props.indstep_proof
simple_props.simple_sensor_step_proof
simple_props.simple_step_lemma_proof
simple_props.stay_simple_proof
simple_props_tcc.simple_sensor_step_lemma_TCC1_PROOF
simple_props_tcc.simple_step_lemma_TCC1_PROOF
supports.Gbar_support_prf
supports.commit_Gbar_lemma_proof
supports.commit_support_proof
supports.commit_when_proof
supports.committed_proof
supports.discharge_antisymmetry
supports.discharge_dichotomy
supports.discharge_reflexive
supports.discharge_transitive
```

```
  supports.found_sub_support_proof
  supports.found_support_proof
  supports.in_own_support_proof
  supports.subset_support_proof
  supports.support_backup_proof
  supports_tcc_proofs.foundation_TCC1_PROOF
  voted_step.main_vote_proof
  voted_step.maj_proof
  voted_step.rrunto_task_proof
  voted_step.rstep_task_proof
  voted_step.sensors_not_voted_proof
  voted_step.sstep_task_proof
  voted_step.vote_lemma_proof
  voted_step.voted_ind_step_proof
  voted_step.voted_link_proof
  voted_step.voted_step_proof
  voted_step_tcc.active_task_TCC1_PROOF
  voted_step_tcc_proofs.rstep_task_TCC1_PROOF
  voted_step_tcc_proofs.sstep_task_lemma_TCC1_PROOF
  voted_step_tcc_proofs.voted_step_lemma_TCC1_PROOF
Total: 93
```

National Aeronautics and Space Administration

Report Documentation Page

1. Report No.
NASA CR-4384

2. Government Accession No.

3. Recipient's Catalog No.

4. Title and Subtitle
Formal Specification and Verification of a Fault-Masking and Transient-Recovery Model for Digital Flight-Control Systems

5. Report Date
July 1991

6. Performing Organization Code

7. Author(s)
John Rushby

8. Performing Organization Report No.

10. Work Unit No.
505-64-10-05

9. Performing Organization Name and Address
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025

11. Contract or Grant No.
NAS1-18969

13. Type of Report and Period Covered
Contractor Report

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

14. Sponsoring Agency Code

15. Supplementary Notes
Langley Technical Monitor: Ricky W. Butler
Task 1 Final Report

16. Abstract

We present the formal specification and mechanically checked verification for a model of fault-masking and transient-recovery among the replicated computers of digital flight-control systems. The verification establishes, subject to certain carefully stated assumptions, that faults among the component computers will be masked--so that commands sent to the actuators will be the same as those that would be sent by a single computer that suffers no failures.

17. Key Words (Suggested by Author(s))
Digital Flight Control Systems, Formal Methods, Formal Specification and Verification, Proof Checking, Fault Tolerance, Transient Faults, Majority Voting, Modular Redundancy

18. Distribution Statement
Unclassified - Unlimited
Subject Category 62

19. Security Classif. (of this report)
Unclassified

20. Security Classif. (of this page)
Unclassified

21. No. of pages
126

22. Price
A07

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171

NASA-Langley, 1991