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

Reprint from J. Vytopil, editor: *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Nijmegen, The Netherlands, January 1992, pages 237–257; Volume 571 of Springer Verlag *Lecture Notes in Computer Science*. Also available as Chapter 5 of J. Vytopil, editor: *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Kluwer International Series in Engineering and Computer Science, 1993.

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Abstract

We present a formal model for fault-masking and transient-recovery among the replicated computers of digital flight-control systems. We establish conditions under which majority voting causes the same commands to be sent to the actuators as those that would be sent by a single computer that suffers no failures. The model and its analysis have been subjected to formal specification and mechanically checked verification using the EHDM system.

Keywords: digital flight control systems, formal methods, formal specification and verification, proof checking, fault tolerance, transient faults, majority voting, modular redundancy

*This research was supported by NASA Langley Research Center under contract NAS1 18969

1 Introduction

Many modern airplanes and spacecraft are crucially dependent on digital flight control systems (DFCS), and so extreme reliabilities are required of those systems. The reliabilities required are beyond those that can be guaranteed for the individual digital devices that constitute the DFCS hardware and so 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 [3]. 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 before being sent to the actuators. 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 to the greatest extent possible. In order that voting should not become a single point of failure, each channel usually has its own voter, and the voted values from each channel are then further voted or averaged at 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 [6, Figure 3.2–2].

Notice that although this approach provides protection against random hardware failures, 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 paper, 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. Redundancy management presents a major challenge in the design of a fault-tolerant N-plex for DFCS. Instead of a single computer executing the DFCS software there will be several—which must coordinate and vote (or average) actuator commands, and tolerate faults among their own members. In addition to the replicated computers, sensors will be replicated also, and their values must be distributed in a Byzantine fault-tolerant manner [14]. 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. Indeed, there is evidence that redundancy management is sufficiently complex and difficult that it can become the *primary* source of unreliability in a DFCS [12, pp. 40–41].

Consequently, the overall goal of a research program led by NASA Langley Research Center, of which this work forms a part, is 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, 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.

The approach we are following, in common with most of those performing research in fault-tolerant systems for DFCS [7,9,11,23], employs synchronized channels and exact-match voting; it also has much in common with the "state-machine" approach [20]. 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. Byzantine fault-tolerant algorithms are also required to synchronize the separate channels. In this way, every (working) computer begins each frame at approximately the same time as the others, and with the same set of sensor readings. Each computer will then run the same sensor selection and averaging algorithms, 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.[1]

We and our colleagues at NASA and the other companies engaged in this program have been undertaking formal specification and verification of some of the key algorithms required to support the synchronized, exact-match voting approach to DFCS. These include Byzantine fault-tolerant clock synchronization algorithms [17, 21] and a Byzantine agreement algorithm [1] and circuits [2, 22]. The work described in this paper is a step towards the next higher layer in the modeling hierarchy: the layer that uses exact-match voting to provide fault-tolerance and

[1] A plausible and apparently simpler approach to redundancy management in an N-plex is the "asynchronous" design, in which the computers run fairly independently of each other: each computer samples sensors and evaluates the control laws independently. The triplex-redundant DFCS of the experimental AFTI-F16 was built this way, and its flight tests revealed some of the shortcomings of the approach [8, 12] (see [16, Chapter 1] for a summary). Asynchronous systems cannot distinguish accurately between the failure of a sensor and that of a computer, and may mistake the consequences of clock drift for either. Nonetheless, aircraft manufacturers continue to use this approach.

transient-recovery. The model and results developed in the following sections have been formally specified and verified using our EHDM system [19].

2 Fault Masking and Transient Recovery in DFCS

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 possible. Examples of transient faults include "single event upsets" (SEUs), where a single bit of memory is flipped by a cosmic ray. These 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 and 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. Thus, reliability analysis must consider the arrival and repair rates of control faults (which are not considered here), and the recovery process for state data faults.

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): 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 paper we 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 [4].

3 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. This model is deliberately simple and abstract. It treats task executions as atomic actions and does not consider their duration in real time. Nor does it model the cyclic, structured pattern of task executions found in statically-scheduled frame-based systems. We exclude real time and assume atomicity (and later perfect synchronization) for simplicity: we prefer to contemplate one issue at a time and to add complexity in stages. We abstract away the concrete details of frame-based organization since they seem irrelevant to (indeed, limitations on) the results we seek. Our colleagues at NASA Langley have formulated a frame-based model [4, 5]; the reconciliation between their model and ours is described in [16, Chapter 4].

3.1 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*; 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.

We assume

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

and we define

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

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

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

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 of active-task cells is denoted C_T and given by $C_T \stackrel{\text{def}}{=} C \setminus C_S$ (where $\setminus$ denotes set difference).

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* $\mathcal{S} \subseteq C \to D$.

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.)

Formalizing the notion of sequential execution, we introduce

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

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 $T \subseteq \mathcal{S} \to D$ of *task-functions*, and

- A function $task\colon C_T \rightarrow T$.

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 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:

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

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.

[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

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

[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 $n = indegree(c)$.

Formally, the execution of cells is modeled by the function

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

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

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

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.

3.2 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

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

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

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

where

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

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:

[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.

[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*.

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

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

(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., $sstep(\rho, c)(i)$. 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 for that cell. This is specified by the function

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

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.

[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.

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]

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

is given by

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

and

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

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.

3.3 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

[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.

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-value*: $R \times C \rightarrow \{T, F\}$

where

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

We then seek a predicate

- *safe*: $C \rightarrow \{T, F\}$

such that

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

Intuitively, $\textit{safe}(c)$ 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: $\textit{safe}(c)$ 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. 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 necessary 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. Notice that particular voting strategies can be modeled within our framework by introducing additional voted tasks whose only function is to gather data for voting. For example "frame-based voting" (voting all the results computed in a given frame at the end of the frame) can be modeled by introducing an additional "voter" task scheduled at the end of each frame. All other tasks in the frame feed copies of their outputs into the voter task (which is the only voted task within the frame), and all subsequent tasks obtain their values from the voter task.

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.

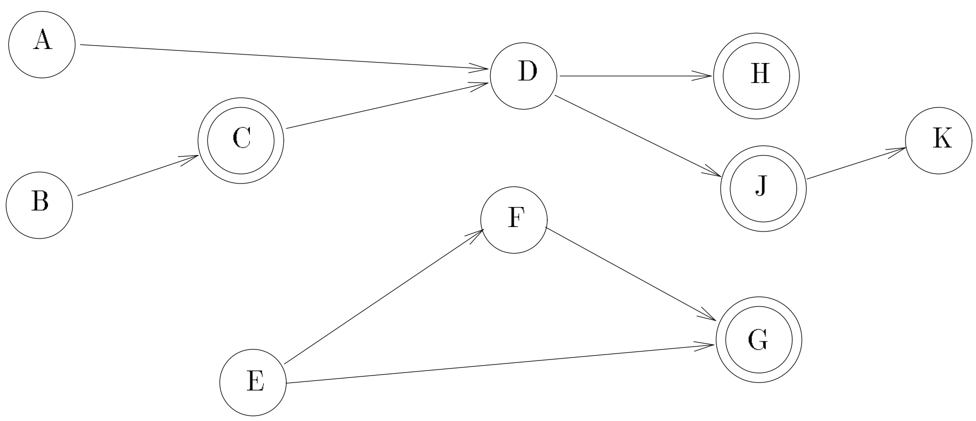

Figure 1: Example Dataflow Dependency Graph

Figure 1 gives a graphical representation of these concepts. In the figure, circles indicate cells, double circles indicate voted cells and the arrows indicate the flow of data (the arrow from cell A to cell D represents the arc $(A, D) \in \overline{G}$). 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 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 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

[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.

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

- $\mathit{committed\text{-}to}: C \rightarrow M$

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 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 from either sensor samples or voted values that are acquired at or later than $\mathit{committed\text{-}to}(c)$. Thus, provided enough other machines 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,

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

is defined by

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

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

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

(for Majority OK) defined as follows

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

We then define the predicate *safe* as follows

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

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 [4].

Theorem 1 (Consensus Theorem) If $\mathit{safe}(c)$, then

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

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

$$rrunto(c)(j)(c) = sensor(c)(1) = runto(c)(c)$$

as required.

For the inductive step, suppose the theorem true for all cells a such that $when(a) < when(c)$ 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

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

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, $\mathit{good\text{-}value}(k, a)$ 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., $\mathit{good\text{-}value}(j, a)$, 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 $\mathit{committed\text{-}to}(c)$ 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 $\mathit{committed\text{-}to}(a)$ can be no earlier than $\mathit{committed\text{-}to}(c)$. 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.

4 Formal Specification and Verification

We have formally specified the model of the previous section in the language of EHDM [19], and formally verified the Consensus Theorem and its Corollary [16]. The specification language of EHDM is a simply-typed higher-order logic with fairly rich facilities for subtyping, dependent typing, modularity, and parameterization. The heart of the formal specification and verification is structured into 11 EHDM modules; in addition, 5 standard modules are used to specify sets, Noetherian induction, and other supporting concepts. The mechanically-checked proof of the Consensus Theorem and its Corollary is structured into 93 elementary lemmas, which can be checked in a total of 7 minutes on a Sun Sparcstation 2. The formal verification rests on a very simple axiomatic basis in addition to the theories (which include arithmetic) built in to the EHDM system itself: four axioms are required in the development of a theory of sets, one is required to state Noetherian induction, and 11 are used to develop the fault masking model. All other formulas are either definitions (these provide conservative extension in EHDM; 15 are used in the specification), or lemmas whose proofs have been checked mechanically.

The model of Section 3 was developed with specification in EHDM in mind; it is built from straightforward mathematical concepts and could therefore be transliterated more or less directly into EHDM. An example module is shown in the appendix (the typesetting was done automatically by EHDM); the formula called `the_result` corresponds to the Consensus Theorem in Section 3. Whereas the formal specification required only a few hours to construct, the formal verification required about three weeks of part-time work. Some of this time was required because the formal verification proves a number of subsidiary results that are glossed over in the ordinary mathematical presentation, and some of it was required because EHDM's proof support was not very efficient at the time the proof was performed (in particular, it lacked a rewriter). We regard the time taken for the verification to be excessive; three days would seem a more reasonable target, and we believe that this will be feasible in the next release of EHDM.

Even three days may seem excessive to some who observe that the proofs given in Section 3 require little more than a page to describe. It should be noted, however, that that proof itself took several days to develop and that it glosses over a number of points that are quite tricky when considered in detail. For example, it assumes that if the output of cell 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 (possibly 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). A rigorous demonstration of this property requires a proof by induction comparable to that required for the main theorem.

Finally, the mechanically verified theorem is stronger than that stated and proved in Section 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 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 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* (and hence obtained a weaker theorem) in the traditional mathematical presentation than we did in the formal specification and verification is that the stronger definition allows the Consensus Theorem 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.

5 Conclusion

We have presented a formal model for fault-masking and transient-recovery among the redundant computers of a DFCS, and proved a condition on the dataflow dependencies in the task graph, the voting pattern, and the arrival of faults, which is sufficient to ensure that correct values are always in the majority among those sent to the actuators. The model and its theorem have been subjected to mechanically-checked specification and verification.

Unlike the verifications of clock synchronization algorithms [17,21], formal verification of our fault-masking model did not uncover flaws or errors in the conventional mathematical presentation. (We attribute this partly to the fact that analysis of the clock synchronization algorithms is very much more difficult, and partly to the fact that considerable experience in formal verification has given us a much refined appreciation whether an *informal* proof is valid.) Formal verification did provide a number of benefits, however: it forced us to attend to a number of details that were glossed over in Section 3, it precisely identified the axiomatic basis of the model, and it gave us a stronger theorem. Identification of the axiomatic basis is important, since it provides the requirements specification for the next lower level in the modeling hierarchy. It also allows us to identify, and we hope later to explore, some fundamental assumptions. For example, plurality voting works just as well as majority voting when a majority exists; we are now in a position to examine the precise properties required of the voting function.

The benefits of formal verification are bought at a price; we believe that price can be reduced substantially in the very near future with carefully-tuned theorem proving support (this is not the same as more powerful theorem proving support). Formal specification in a suitably rich specification language, however, exacts very little price at all, and exerts a beneficial discipline. We see no reason why it should not be routine. Indeed, the main point we wish to make is that modern verification systems can support a sufficiently rich fragment of mathematics that it is feasible to develop models of important elements of interesting computer systems directly in that mechanized fragment of mathematics, without compromising convenience or elegance of expression. We analyze the attributes that maximize convenience and utility of specification languages and verification systems elsewhere [15, 18].

Our plans for the future include elimination of some of the simplifying assumptions of the present model. Accurate modeling of fault masking must recognize 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. 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 [10, 11] is a good example of a system that provides such recovery. Our plans for the future also include formal analysis of selected mechanisms for recovering from transient control faults. In other current work, we are developing and formally verifying a hardware-assisted implementation of one of the clock-synchronization algorithms.

Summarizing the results of the AFTI-F16 flight tests, NASA engineer Mackall observed [12, pp. 40–41]

> "...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."

We hope that the work described here (and our larger program) contributes to the goal of developing testable designs, purged of "random characteristics," and which satisfy Mackall's plea for a method that will make designs "understandable, more visible."

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 paper 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] W. R. Bevier and W. D. Young. Machine-checked proofs of a Byzantine agreement algorithm. Technical Report 55, Computational Logic Incorporated, Austin, TX, June 1990.

[2] W. R. Bevier and W. D. Young. The design and proof of correctness of a fault-tolerant circuit. In Meyer and Schlichting [13], pages 243–260.

[3] Ricky W. Butler and Sally C. Johnson. The art of fault-tolerant system reliability modeling. NASA Technical Memorandum 102623, NASA Langley Research Center, Hampton, VA, March 1990.

[4] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell. Formal design and verification of a reliable computing platform for real-time control. NASA Technical Memorandum 102716, NASA Langley Research Center, Hampton, VA, October 1990.

[5] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell. High level design proof of a reliable computing platform. In Meyer and Schlichting [13], pages 279–306.

[6] Carl S. Droste and James E. Walker. *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.

[7] Richard E. Harper and Jaynarayan H. Lala. Fault-tolerant parallel processor. *AIAA Journal of Guidance, Control, and Dynamics*, 14(3):554–563, May–June 1991.

[8] Stephen D. Ishmael, Victoria A. Regenie, and Dale A. Mackall. Design implications from AFTI/F16 flight test. NASA Technical Memorandum 86026, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, CA, 1984.

[9] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Transactions on Computers*, 37(4):398–405, April 1988.

[10] H. Kopetz, H. Kantz, G. Grünsteidl, P. Puschner, and J. Reisinger. Tolerating transient faults in MARS. In *Fault Tolerant Computing Symposium 20*, pages 466–473, Newcastle upon Tyne, UK, June 1990. IEEE Computer Society.

[11] Hermann Kopetz et al. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, 9(1):25–40, February 1989.

[12] Dale A. Mackall. Development and flight test experiences with a flight-crucial digital control system. NASA Technical Paper 2857, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, CA, 1988.

[13] J. F. Meyer and R. D. Schlichting, editors. *Dependable Computing for Critical Applications—2*, volume 6 of *Dependable Computing and Fault-Tolerant Systems*, Tucson, AZ, February 1991. Springer-Verlag, Wien, Austria.

[14] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[15] John Rushby. Design choices in specification languages and verification systems. In Phillip Windley, editor, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 195–204, Davis, CA, August 1991. IEEE Computer Society.

[16] John Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. Technical Report SRI-CSL-91-3, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1991. Also available as NASA Contractor Report 4384, July 1991.

[17] John Rushby and Friedrich von Henke. Formal verification of the Interactive Convergence clock synchronization algorithm using EHDM. Technical Report SRI-CSL-89-3R, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1989 (Revised August 1991). Original version also available as NASA Contractor Report 4239, June 1989.

[18] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[19] John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.

[20] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[21] Natarajan Shankar. Mechanical verification of a schematic Byzantine fault-tolerant clock synchronization algorithm. Technical Report SRI-CSL-91-4, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1991. Also available as NASA Contractor Report 4386, July 1991.

[22] Mandayam Srivas and Mark Bickford. Verification of the FtCayuga fault-tolerant microprocessor system, volume 1: A case-study in theorem prover-based verification. Contractor Report 4381, NASA Langley Research Center, Hampton, VA, July 1991. (Work performed by ORA Corporation).

[23] John H. Wensley et al. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

A Example Specification Module

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$[R \to \text{set}[C]] =$
$(\lambda\, i :$
$(\lambda\, c :$
$(\forall\, m : \text{committed\_to}(c) \leq m \wedge m \leq \text{when}(c) \supset \neg F(i)(m))))$

working: function$[C \to \text{set}[R]] == (\lambda\, c : (\lambda\, i : \text{OK}(i)(c)))$

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

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

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

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

End correctness