

# Assured Reconfiguration of Fail-Stop Systems

Elisabeth A. Strunk      John C. Knight      M. Anthony Aiello

*Department of Computer Science, University of Virginia  
151 Engineer's Way, Charlottesville, VA 22904-4740  
{strunk | knight | aiello}@cs.virginia.edu*

## Abstract

*Hardware dependability improvements have led to a situation in which it is sometimes unnecessary to employ extensive hardware replication to mask hardware faults. Expanding upon our previous work on assured reconfiguration for single processes and building upon the fail-stop model of processor behavior, we define a framework that provides assured reconfiguration for concurrent software. This framework can provide high dependability with lower space, power, and weight requirements than systems that replicate hardware to mask all anticipated faults. We base our assurance argument on a proof structure that extends the proofs for the single-application case and includes the fail-stop model of processor behavior. To assess the feasibility of instantiating our framework, we have implemented a hypothetical avionics system that is representative of what might be found on an unmanned aerial vehicle.*

## 1. Introduction

Schlichting and Schneider introduced the concept of fail-stop processors as a building block for safety-critical systems, and they presented a programming approach based on fault-tolerant actions (FTAs) in which software design takes advantage of fail-stop semantics [8]. Their approach enables the construction of dependable logical machines composed of less dependable physical components. Since their original work, the dependability of off-the-shelf hardware components has risen substantially; also, in many systems, weight and power are still limiting factors, even though capability has increased significantly. In this paper, we extend fail-stop processors with reconfiguration semantics so that they can contribute to system dependability without adding extra hardware. We do this by: (1) allowing the recovery protocol to specify that the system will reconfigure rather than complete the action; and (2) allowing FTAs to span multiple

applications so that faults which would normally have to be masked can be dealt with by changing the functionality of the system as a whole. System service is restricted during reconfiguration, although briefly.

System reconfiguration is currently used for a variety of purposes in safety-critical systems: for example, to effect changes in functionality between different mission phases for spacecraft or between different operating modes for aircraft. Also, although the vast majority of equipment failures in safety-critical systems are dealt with by masking their effects using replicated components, reconfiguration is sometimes used to cope with the failure of specialized equipment such as sensors. In addition, it can be used to cope with the failure of computing equipment, as in the Boeing 777, where replication is used to mask many effects of computer and data bus failures, but where the system can be reconfigured to provide reduced functionality if more than the expected number of failures occurs [12].

We claim that reconfiguration of a system built with fail-stop processors can be used effectively to tolerate many faults in dependable systems that would be expensive or impossible to deal with through replication alone, and that it can become the heart of the architecture of a safety-critical system. If reconfiguration is to be given a central role in system dependability, its assurance becomes paramount. The reconfiguration protocols currently used in practice are system-specific and are built, in large measure, using whatever architectural facilities are already provided by the system. In an earlier paper [10], we presented an approach to assured reconfiguration for a single application process that consisted of multiple modules. In this paper, we extend our previous work, introducing an approach to the assured reconfiguration of a set of application processes. To this end, we present a system architecture and verification framework in which the reconfiguration logic is a customizable mechanism that will: (1) accept component-failure signals; (2) determine the configuration to which the system should move; and (3) send configuration signals to the individual pro-

cesses to cause them to respond properly to component failure. To meet our assurance goal, we have shown that our architecture satisfies a generalized version of the single-application properties.

To illustrate the ideas that we describe, we have built part of a hypothetical avionics system that is typical of what might be found on a modern general-aviation aircraft or an unmanned aerial vehicle (UAV). The system includes a flight control application, an electrical power generation monitoring application, and an autopilot. The parts of these applications that are relevant to our research have been implemented, although the functionality is merely representative.

In section 2 we review related work, and in section 3 we introduce our system architecture. Assumptions are listed section 4. We review Schlichting and Schneider's work in section 5. In section 6 we present our formal model of reconfiguration. Our example avionics system is described in section 7. Section 8 concludes.

## 2. Related work

Other researchers have proposed the use of reconfiguration to increase system dependability in a variety of contexts. Shelton and Koopman have studied the identification and application of useful alternative functionalities that a system might provide in the event of hardware component failure [9]. Their work is focused, however, more on reconfiguration requirements than effecting the reconfigurations themselves. Sha has studied the implementation of reconfiguration in fault tolerance for control systems [7], although his work does not focus on assurance. Likewise, Garlan et al. [2] have proposed the use of software architectural styles as a general method of error detection and reconfigura-

tion execution to improve dependability, but they do not present a method of assuring their styles.

In large networked systems, reconfiguration in response to failures is known as *information system survivability*. Informally, a survivable system is one that has facilities to provide one or more alternative services (degraded, less dependable, or otherwise different) in a given operating environment [4]. For networked systems, the loss of a single component or even a moderate number of randomly-distributed components must be expected. System reconfiguration is employed only in the event of damage with significant consequences, or if moderate numbers of failures suggest a common cause. The main challenge of reconfiguration in these systems is managing system scale.

Our work is part of a framework for using reconfiguration in embedded real-time systems. The embedded system reconfiguration requirements are similar to the networked system requirements, with three key differences: (1) the system is much smaller than large-scale information systems and thus can be tightly controlled; (2) the system might be expected to respond much more quickly to a failure and thus have hard real-time reconfiguration requirements; and (3) a failure of any application to carry out a reconfiguration can have a much greater impact on the system, and so the assurance requirements of a functional transition are much more demanding.

## 3. System architecture overview

To introduce the elements of our architecture and show how they fit together, we begin with an overview, illustrated in Figure 1. The architecture assumes a distributed computing platform consisting of an unspeci-

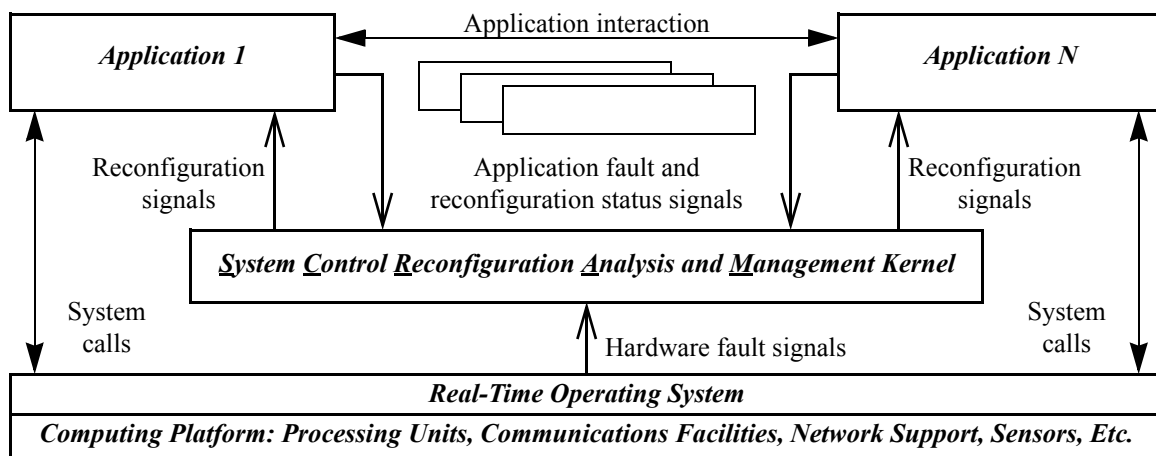


Figure 1. Logical System Architecture

fied number of processing elements that communicate via an ultra-dependable, real-time data bus. Each processing element consists of a fail-stop processor with associated volatile and stable storage that executes a real-time operating system. An example fail-stop processor might be a self-checking pair; an example data bus might be one based on the time-triggered architecture [5]; and an example operating system might be one that complies with the ARINC 653 specification [1]. Sensors and actuators that are used in typical control applications are connected to the data bus via interface units that employ the communications protocol required by the data bus.

The system that the architecture supports consists of a set of applications, each of which operates as an independent process with no assumptions on how processes are mapped to platform nodes except that the mapping is statically determined. Applications communicate via message passing or by sharing state through the processors' stable storage.

Each application implements a set of specifications and provides an interface for *internal* reconfiguration [6]. Each specification for each application is defined by domain experts; certain specification combinations, denoted *configurations* and defined in a *reconfiguration specification* [11], provide acceptable services. Reconfiguration is the transition from one configuration to another. Reconfiguration and its assurance are the heart of our architecture.

System reconfiguration is effected by the System Control Reconfiguration Analysis and Management (SCRAM) kernel. This kernel implements the *external* reconfiguration [6] portion of the architecture by receiving component failure signals when they occur and determining necessary reconfiguration actions based on a statically-defined set of valid system transitions. Component failures are detected by conventional means such as activity, timing, and signal monitors. A detected component failure is communicated to the SCRAM via an abstract signal, and the kernel effects reconfiguration by sending sequences of messages to each application's reconfiguration interface.

The reconfiguration message sequence causes operating applications to stop executing their current specification, establish a postcondition from which a new specification can be started, and initiate operation in a pre-determined new specification. Each application meets prescribed time bounds for each stage of the reconfiguration activity, thereby ensuring that reconfiguration is always completed in bounded time.

The time and service guarantees that our architecture provides hinge on the correct and timely operation of the SCRAM. A dependable implementation of this

function could be created in various ways, such as distributing it over multiple processors and protecting it against failure of a subset of those processors, or allocating it to a fail-stop processor so that any faults in its hardware will be masked. We do not address the specifics of the SCRAM implementation in this paper.

## 4. System assumptions

We state here the primary assumptions we make on system requirements. These assumptions are those necessary for our architecture to be applicable to a system; additional assumptions made for simplicity are addressed as needed throughout the rest of the paper.

First, we assume that a system is synchronous and made up of a set of applications  $Apps = \{a_1, a_2, \dots, a_m\}$ , which may run on an arbitrary processor configuration, and which may interact. Each  $a_i$  in  $Apps$  possesses a set of possible functional specifications  $S_i = \{s_{i1}, s_{i2}, \dots, s_{in}\}$  and always operates in accordance with one of those specifications unless engaged in reconfiguration. Any functional dependencies among the applications in  $Apps$  must be acyclic.

Second, we assume that it is possible to know in advance all of the desired potential system configurations  $C = \{c_1, c_2, \dots, c_p\}$  and how to choose between them. The system will have at least one "safe" configuration, which is built with high enough dependability that failures at the rate anticipated for the safe configuration do not compromise system dependability goals.

Third, we assume that system function can be restricted while the system is reconfigured. We discuss time bounds on function restriction in section 5.3.

Finally, we assume fail-stop computers and standard hardware error detection (and, in some cases, masking) mechanisms for other system elements. We assume the existence of a reconfiguration trigger; the source of the trigger might be a hardware failure, a software functional failure, the failure of software to meet its timing constraints, or a change in the external environment that necessitates reconfiguration but involves no failure at all. We do not address error detection mechanisms for any system component.

## 5. Fail-stop processors and fail-stop processes

Fundamental to the thesis of this paper is that component failure occurs in safety-critical systems and need not be masked, provided the system can be made reconfigurable. To have confidence in the function and

dependability of a reconfigurable architecture, complete definitions of the failure semantics of all components that are expected to fail must be provided. The most desirable semantics are those referred to as *fail-stop*. In this section, we review the concept underlying fail-stop processors, discuss their use, and summarize our approach to reconfiguration using them.

## 5.1 Fail-stop processors

Schlichting and Schneider define a fail-stop processor as consisting of one or more processing units, volatile storage, and stable storage [8]. A fail-stop processor's failure semantics can be summarized as:

- The processor stops executing at the end of the last instruction that it completed successfully.
- The contents of volatile storage are lost, but the contents of stable storage are preserved.

An embedded system of the type discussed in this paper is made up of a collection of fail-stop processors. If one processor fails, the others poll its stable storage to find out what state it was in when it failed. If necessary, the system then reconfigures to meet its reconfiguration specification. In a system where faults are masked (as is assumed by Schlichting and Schneider), there must be sufficient equipment available to provide full service if the anticipated number of component failures occurs during the maximum planned mission time. The total number of required components is thus the sum of the maximum number expected to fail during the longest planned mission and the minimum number needed to provide *full* service. Though not impossible, loss of the maximum number expected to fail is an *extremely* unlikely occurrence. Thus, the vast majority of the time, the system will be operating with far more computing resources than it needs.

With the approach we advocate, the total number of required components is the sum of the maximum number expected to fail during the longest planned mission and the minimum number needed to provide the most *basic* form of safe service. If the system were designed so that this number equals the number of components needed to provide full service, then, during routine operation (i.e., the vast majority of the time), the system would operate with no excess equipment. Even if some system functions do not meet the assumptions listed in the previous section, failures of those functions can be masked, while failures in other functions can trigger a reconfiguration. Reconfiguration in place of masking, or the combination of reconfiguration with masking, saves power, weight, and space.

## 5.2 Software considerations in fail-stop processors

Schlichting and Schneider introduced the concept of a *fault-tolerant action* (FTA) as a building block for programming systems of fail-stop processors. Briefly, an FTA is a software operation that either: (1) completes a correctly-executed action  $A$  on a functioning processor; or (2) experiences a hardware failure that precludes the completion of  $A$  and, when restarted on another processor, completes a specified recovery action  $R$ . Thus, an FTA is composed of either a single action, or an action and a number of recoveries equal to the number of failures experienced during the FTA's execution. Using FTAs, Schlichting and Schneider show how application software can be constructed so as to mask the functional effects of a fail-stop processor failure and how proofs can be constructed to show that state is properly maintained.

In the original framework, a recovery protocol may complete only the original action, either by restarting it or by some alternative means. Our framework takes a broader view of the recovery protocol, where  $R$  might be the reconfiguration of the system so that the next  $A$  will complete some useful but *different* function. An FTA in our framework, then, leaves the system either having carried out the function requested, or having put itself into a state where the next action can carry out some suitable but possibly different function.

In the approach presented here, the basic software building block is a *reconfigurable application*. Where the meaning is clear, we refer to a reconfigurable application as an application in the remainder of this paper. An application has a predetermined set of specifications with which it can comply and, correspondingly, a predetermined set of fault-tolerant actions that are appropriate under each specification. Which recovery protocol is appropriate for use when an application fails, however, cannot be determined by the application alone since the application's function exists in a system context. Furthermore, applications may depend on one another, so that the initial failure of an action in one application could lead to the failure of an action in another application. This issue did not arise in the previous formulation of fail stop since failures were completely masked; with the possibility of reconfiguration, however, a distinction must be drawn between *application* FTAs (AFTAs) and *system* FTAs (SFTAs).

An AFTA is an action encompassing a single unit of work for an individual application. An SFTA is composed of a set of AFTAs. Because of system synchrony, there is some time span in which each application will have executed a fixed number of AFTAs. The AFTAs

that are executed during that time span make up the SFTA. If an application experiences a failure but recovers from that failure without affecting any other applications, then the SFTA includes that application's action and subsequent recovery, as well as the standard AFTAs for the other applications.

Because an application failure can affect other applications, some mechanism for determining what other application reconfigurations are necessary to complete an SFTA is required. It is for this purpose that we introduce the System Control Reconfiguration Analysis and Management (SCRAM) kernel. Possible configurations to which the system might move to complete its SFTA are statically defined as set of valid system transitions and a function to determine which transitions must be taken under each possible set of operating circumstances. The SCRAM signals the remaining applications to effect the appropriate reconfiguration.

A software system composed of reconfigurable applications can be reconfigured to meet a given system specification, provided appropriate configurations exist for all the applications. Transition existence can be guaranteed in a straightforward way by including a coverage requirement over environmental transitions, potential failures, and permissible reconfigurations.

Applications lost due to a processor failure are known to have been lost because of the static association of applications to processors. We assume nothing about the state of an application when it fails.

### 5.3 Real-time behavior of fail-stop software

We define a reconfigurable application to have the following informal properties:

- The application responds to an external halt signal by establishing a prescribed postcondition and halting in bounded time.
- The application responds to an external reconfiguration signal by establishing the precondition necessary for the new configuration in bounded time.
- The application responds to an external start signal by starting operation in whatever configuration it has been assigned in bounded time.

We model time at the system level by associating uniform timing bounds with each stage of each AFTA. Because of the precise semantics of an SFTA, timing guarantees can be given for SFTAs based on the bounds provided for AFTAs for a reconfiguration that is able to complete with no intervening failures.

While Schlichting and Schneider's work includes a discussion of system temporal properties in the event of multiple successive failures, including how one

might guarantee liveness, we extend the framework to cover reconfiguration timing for systems that must operate in hard real time. Any failures that occur during reconfiguration can be either (1) addressed immediately by ensuring the applications have met their postconditions and choosing a different target specification; or (2) buffered until the next stable storage commit of other applications, depending on system requirements. In the worst case, each failure cannot be dealt with until the end of the current reconfiguration. In this case, the longest restriction of system function is equal to the sum of the maximum time allowed between each reconfiguration in the longest chain of transitions to some safe system configuration  $C_s$ . In other words, for the longest configuration chain  $C_1, C_2, \dots, C_s$ , the maximum restriction time is  $\sum_{i=2 \text{ to } s} T_{i-1, i}$ , where  $T_{i, j}$  is the maximum time to transition from  $C_i$  to  $C_j$ . This time can be reduced in various ways, such as interposing a safe configuration  $C_s$  in between any transition between two unsafe configurations. With this addition, the new maximum time over all possible system transitions  $C_i \rightarrow C_j$  would be  $\max\{T_{i, s}\}$ .

One caveat of this formula is that cyclic reconfiguration is possible due to repeated failure and repair or rapidly-changing environmental conditions, and in this case the time to reconfigure could be infinite. Potential cycles can be detected through a static analysis of permissible transitions. They can be dealt with by forcing a check that the system has been functional for the necessary amount of time (in a safe configuration, or in a configuration appropriate to all environmental conditions) before a subsequent reconfiguration takes place.

## 6. Formal model

The above discussion gives an overview of how we have extended Schlichting and Schneider's work to support reconfiguration, but, as an informal discussion, it lacks the rigor necessary for assurance of dependable systems. To provide this assurance, we have created an assurance argument that is an extension of that presented in our previous work [10]. The assurance argument includes: (1) a formal model of a reconfigurable system architecture; (2) a set of formal properties, stated as putative theorems over the model, that we use as our definition of system reconfiguration; and (3) proofs of the theorems—which constitute a proof that the architecture satisfies the definition. With this verification framework in place, we know that any instance of that architecture will be a valid instance of a recon-

figurable system as long as the proof obligations generated by the formal type system have been met.

Our formal model is specified in PVS, and proofs of the putative theorems have been mechanically verified with the PVS system. We have also formally specified the essential interfaces of an example reconfigurable system (see section 7) and shown that this example has the necessary properties of our architecture. The result is an assurance argument based on proof. In this section, we present the model and its properties.

## 6.1 Model assumptions

Assuring real-time properties is difficult in general; although PVS is a powerful specification and proving system, real-time properties of complex systems can be nontrivial. In this work, in order to enable the specification and proof structure that we seek, we make the following general assumptions on applicable systems:

- Each application operates with synchronous, cyclic processing and with a fixed real-time frame length.
- All applications operate with the same real-time frame length.
- The real-time frames for all applications are synchronized to start together.
- Each application completes one unit of work in each real-time frame (where that unit of work can be normal function, halting an application and restoring its state, preparing an application to transition to another specification, or initializing data such as control system gains).
- Each application commits results to stable storage at the end of each computation cycle (real-time frame).
- Any dependencies between applications require only that the independent application be halted before the dependent application computes its precondition.

These assumptions impose restrictions on a system to facilitate analysis. Despite these assumptions, the

model presented here supports the development of a useful class of systems.

## 6.2 Application model

During normal operation, each application reads data values produced by other applications from stable storage at the start of each computational cycle and then performs its computation. It commits its results back to stable storage at the end of each computational cycle. Should reconfiguration become necessary, the AFTAs are not able to execute their recovery protocols immediately or independently because they depend both on their own state and on other system state. Each must wait, therefore, for the SCRAM to coordinate all of the currently executing AFTAs.

The SCRAM communicates with applications through variables in stable storage. When reconfiguration is necessary, it sets the *configuration\_status* variable to a sequence of values on three successive real-time frames. The three values are: *halt*, *prepare*, and *initialize*. At the beginning of each real-time frame, each application reads its *configuration\_status* variable and completes the required action during that frame as shown in Table 1.

## 6.3 System model

The SCRAM effects a system reconfiguration by reconfiguring all of the applications in the system using the sequence of actions shown in Table 1. In the table, application *i* has failed, and the new system configuration will be  $C_t$ . The various sequences are coordinated by the SCRAM to ensure that dependencies among AFTAs are respected.

This approach differs slightly from the approach taken by Schlichting and Schneider. In their approach, system processing is achieved by the execution of a sequence of FTAs where the recovery for a given FTA

**Table 1: SFTA Phases**

Frame	Message	Action	Predicate
0 (start)	Application <i>i</i> : failure signal → SCRAM SCRAM: halt → all apps	None	None
1	None	Applications cease execution	Application postconditions
2	SCRAM: prepare( $C_t$ ) → all apps	Applications prepare to transition to $C_t$	Application transition conditions for $C_t$
3 (end)	SCRAM: initialize → all apps	Applications initialize, establish operating state for $C_t$	Application preconditions for $C_t$

is executed after the associated action if it is interrupted. In our approach, the recovery for a system fault-tolerant action that is interrupted consists of three parts: (Frame 1) each executing AFTA establishes a required postcondition and reaches a halted state; (Frame 2) each executing AFTA establishes the condition to transition to operation under the new state; and (Frame 3) each executing AFTA establishes its precondition: all state associated with the AFTA has been initialized, and the application is functioning normally.

The functional aspects of the SCRAM will remain constant, unless changes are needed because of specific dependencies. This simplifies subsequent verification, since the SCRAM need only be verified once. The SCRAM must, however, be provided with application-specific data that is created and verified separately:

- *An interface to one or more applications that monitor environmental characteristics.* These applications calculate the effective state of the environment.
- *A table of potential configurations.* This defines potential system configurations and maps system configurations into application specifications.
- *A function to choose a new configuration.* The function maps current configuration and environment state to a new configuration. This function implicitly includes information on valid transitions.
- *An interface to each application.* This contains the configuration variable for the application.

Our formal system model includes an overarching function in the PVS specification to coordinate and control application execution. This is a mechanism that is present as part of the PVS structure; in practice, timing analysis and synchronization primitives would be used to achieve frame coordination. Variables in stable storage shared between the SCRAM and the relevant application ensure synchronization during reconfiguration stages; the synchronization, in turn, ensures preservation of data dependencies.

The SCRAM’s synchronization mechanism can be extended to support richer application interdependencies, as long as the dependencies are acyclic and time permits. Given a specification of dependencies, it could preserve them by checking each cycle to see if the independent application has completed its current configuration phase. Only after that phase is complete would the SCRAM signal the dependent application to begin its next stage. Dependencies could also be relaxed by removing any unnecessary intermediate stages or allowing the applications to complete multiple sequential stages without signals from the SCRAM.

Since we make no distinction between failures and other environmental changes, the status of a compo-

nent is modeled as an element of the environment, and a failure is simply a change in the environment. Any environmental factor whose change could necessitate a reconfiguration can have a virtual application to monitor its status and generate a signal if the value changes.

## 6.4 Formal definition of reconfiguration

Using the above model, we now outline the high-level properties we require of any reconfiguration. Defining these properties in an abstract sense allows us to argue that the general requirements needed for assured reconfiguration have been met. The model was constructed to enable proof of these properties.

In previous work, we informally defined reconfiguration of a single application as:

*the process through which a system halts operation under its current source specification  $S_i$  and begins operation under a different target specification  $S_j$  [10].*

For the multiple application case, we define reconfiguration informally as:

*the operation through which a function  $f: Apps \rightarrow S$  of interacting applications  $A$  that operate according to certain specifications in  $S$  transitions to a function  $f': Apps \rightarrow S$  of interacting applications  $Apps$  that operate according to different specifications in  $S$ .*

An SFTA is thus comprised of correct execution of all applications  $a_i$  under their respective specifications  $f(a_i)$ . System reconfiguration is only necessary if  $a_i$  cannot mask the failure, but must transition to an alternative specification in order to complete its AFTA. If only  $a_i$  must reconfigure, then  $\forall a_j \neq a_i, f'(a_j) = f(a_j)$ .

For our system model, including application specification and system configuration types, we created properties that we use to define “correct” reconfiguration, shown in Table 2. To represent them, we include:

```
reconfiguration: TYPE =
  [# start_c: cycle, end_c: cycle #]
  — execution cycle in which the reconfiguration
  starts and ends

sys_trace : TYPE =
  — possible state traces
[# ...
  sp: reconf_spec,
  — sp 'apps is the set of system applications
  tr: [cycle -> s: sys_state],
  — function from system cycle to system state
```

**Table 2: Formal Properties of System Reconfiguration**

Informal Description	Formal Property
<i>SP1: R begins at the same time any application in the system is no longer operating under <math>C_i</math> and ends when all applications are operating under <math>C_j</math></i>	<pre>FORALL (s: sys_trace, r: (get_reconfigs(s))) :   (EXISTS (app: (s`sp`apps)) :     s`tr(r`start_c)`reconf_st(app) = interrupted) AND   (FORALL (app: (s`sp`apps)) :     s`tr(r`start_c-1)`reconf_st(app) = normal) AND   (FORALL (app: (s`sp`apps)) :     s`tr(r`end_c)`reconf_st(app) = normal) AND   (FORALL (c: cycle, app: (s`sp`apps)) :     r`start_c &lt; c AND c &lt; r`end_c =&gt;     s`tr(c)`reconf_st(app) /= normal)</pre>
<i>SP2: <math>C_j</math> is the proper choice for the target system specification at some point during R</i>	<pre>FORALL (s: sys_trace, r: (get_reconfigs(s))) :   EXISTS (c: real_time) :     r`start_c &lt;= c AND c &lt;= r`end_c AND     s`tr(r`end_c)`svclvl =       s`sp`choose(s`tr(r`start_c)`svclvl,         s`env(c*cycle_time))</pre>
<i>SP3: R takes less than or equal to <math>T_{ij}</math> time units</i>	<pre>FORALL (s: sys_trace, r: (get_reconfigs(s))) :   (r`end_c - r`start_c + 1)*cycle_time &lt;=     s`sp`T(s`tr(r`start_c)`svclvl, s`tr(r`end_c)`svclvl)</pre>
<i>SP4: The precondition for <math>C_j</math> is true at the time R ends</i>	<pre>FORALL (s: sys_trace, r: (get_reconfigs(s))) :   pre?(s, r`end_c)</pre>

env: valid\_env\_trace(sp`E, sp`R) #]  
 — given environmental states

get\_reconfigs(s: sys\_trace) :  
 set[reconfiguration]  
 — set of all system reconfigurations

Formally, an SFTA is an action  $R$  in which the conditions in Table 2 hold. We have proven the properties in Table 2 directly from our abstract formal specification using the PVS system. The specification is a set of types that enforces all of the desired system properties by placing type restrictions on any instantiation. In our case, an instantiation of the PVS architecture would be a PVS specification itself. The powerful type mechanisms of PVS are used to automatically generate all of the proof obligations required to verify that a system instance is compliant with the desired properties. In this work, we have proven all of the desired properties of our abstract specification in PVS, and the PVS proof checker has mechanically verified that the proofs are sound (thus we do not include the proofs here).

## 7. Example instantiation

To assess the feasibility of the approach outlined in this paper and to demonstrate the concepts that constitute the approach, we have implemented an example reconfigurable system. The system is a hypothetical

avionics system that is representative, in part, of what might be found on a modern UAV or general-aviation aircraft. The example includes two functional applications: an autopilot and a flight control system (FCS). Only minimal versions of application functionality have been implemented since the system is not intended for operational use. However, each application has a complete reconfiguration interface, including the capability to provide multiple functionalities. An electrical power generation system is modeled as an environmental factor that might necessitate a reconfiguration.

In its primary specification, the autopilot provides four services to aid the pilot: altitude hold, heading hold, climb to altitude, and turn to heading. It also implements a second specification in which it provides altitude hold only. Its second specification requires substantially less processing and memory resources.

The FCS provides a single service in its primary specification: it accepts input from the pilot or autopilot and generates commands for the control surface actuators. This primary specification could include stability augmentation facilities designed to reduce pilot workload, although we merely simulate this. The FCS also implements a second specification in which it provides direct control only, i.e., it applies commands directly to the control surfaces without any augmentation of its input. As with the autopilot, when operating

under this specification, the FCS requires less processing and memory resources.

The electrical system consists of two alternators and a battery, and its interface exports the state that it is in. One alternator provides primary vehicle power; the second is a spare, but normally charges the battery, which is an emergency power source. Loss of one alternator reduces available power below the threshold needed for full operation. Loss of both alternators leaves the battery as the only power source. The electrical system operates independently of the reconfigurable system; it merely provides the system details of its state. For illustration, the anticipated component failures for which reconfiguration takes place are all based on the electrical system.

Our system can operate in three configurations:

- *Full Service.* Full power is available, and all of the platform computing equipment can be used. The autopilot and FCS provide full service, and each operates on a separate computer.
- *Reduced Service.* Power is available from only one alternator or the battery, and some of the platform computing equipment has to be shut down. The applications must share a single computer that does not have the capacity to support full service from the applications, so the autopilot provides altitude hold service only and the FCS provides direct control.
- *Minimal Service.* In this configuration, power is available from the battery only, and the remaining platform computing equipment has to be switched to its low-power operating mode. The applications must share a single computer that is operating in low-power mode, and so the autopilot is turned off and the FCS provides direct control.

The reconfiguration interfaces for the two applications described above, the three acceptable configurations, and the transitions between configurations are specified in PVS. We type checked our instantiation against the abstract specification described in section 6 and discharged the generated proof obligations.

We have constructed a Java implementation of the example, but have not verified it against its specification since the focus of this work is at the specification level. The implementation runs on a set of personal computers running Red Hat Linux. Real-time operation is modeled using a virtual clock that is synchronized to the clocks provided by Linux. A time-triggered, real-time bus and stable storage are simulated. This example has been operated in a simulated environment that includes aircraft state sensors and a simple model of aircraft dynamics. Its potential reconfigurations have been triggered by simulated failures of the electrical

system and executed by the application and SCRAM instantiations.

## 7.1 An example system fault tolerant action

In our example, each AFTA is implemented as described in section 6.2. For illustration, we require certain hypothetical constraints on system transitions that manage aircraft dynamics properly. In any transition, the aircraft's condition in the target configuration must be known so that processing in the target configuration begins correctly. In this example, we require that the control surfaces be *centered*, i.e., not exerting turning forces on the aircraft, and the autopilot be *disengaged* when a new configuration is entered. These are thus the preconditions for the FCS and autopilot. The postcondition that each application must establish prior to reconfiguration is merely to cease operation.

The specific applications that we have used in this illustration have no dependencies during their halt stages because neither requires any support to establish its postcondition. There is only one dependency during initialization, namely that the autopilot cannot resume service in the *Reduced Service* configuration until the FCS has completed its reconfiguration—the autopilot cannot effect control without the other application.

Suppose that the system is operating in the *Full Service* configuration and an alternator fails. The electrical system will switch to use the other alternator, and its interface will inform the SCRAM of the failure. The autopilot and flight control applications are unable to complete their AFTAs within the allotted time because there is not enough power to run them both at full service; however, their postconditions are simply to cease operation, and so this does not pose a problem. Based on the static reconfiguration table, the SCRAM commands a change to the *Reduced Service* configuration. The system reconfiguration is implemented using the sequence shown in Table 1 to complete its SFTA.

## 7.2 Properties of the example instantiation

As discussed above, the formal properties from section 6.4 are enforced in specification instantiations through the type system. PVS does this by generating type-correctness conditions (TCCs), a kind of proof obligation. The example proof obligation in Figure 2, abbreviated and otherwise edited for clarity, requires that: (1) the example's specification levels are of the type expected as the SCRAM input parameters; and (2) the `covering_txns` predicate (which ensures a transition exists for any possible failure-environment pair)

```

% Subtype TCC generated(at line 117, column 16) for ex_SCRAM_table
% expected type SCRAM_table(ex_apps, extend[...] (ex_speclvl)),
% ex_valid_env, ex_reachable_env)
% proved - complete
example_reconf_spec_TCC6: OBLIGATION
FORALL (x: speclvl): ex_speclvl(x) IFF
  extend[speclvl, {sp: speclvl | NOT sp = indeterminate}, bool, FALSE]
  (restrict[speclvl, {sp: speclvl | NOT sp = indeterminate}, boolean] (ex_speclvl)) (x)
AND covering_txns(ex_apps, extend[...] (ex_speclvl)), ex_valid_env, ex_reachable_env,
ex_SCRAM_table`txns, ex_SCRAM_table`primary, ex_SCRAM_table`start_env);

```

**Figure 2. Example TCC**

must hold for the parameters. We have proven all of the TCCs generated for the example.

## 8. Conclusion

Safety-critical systems often use hardware replication to tolerate faults that occur during operation, and Schlichting and Schneider have presented a theory of doing this based on the rigorous semantics of fail-stop processors. Reconfiguring the system in response to faults, however, can help designers achieve their dependability goals without necessitating additional hardware, thus saving weight, power, and space. In this paper, we have drawn on a previous work in reconfiguration of single applications and the semantics presented by Schlichting and Schneider to create an architecture and verification framework to use reconfiguration in dependable systems. We address the requirements of systems of interacting applications, combining the distributed system aspect of fail stop with the structured, proof-based assurance of our previous work to address timing as well as temporal characteristics of systems. This enables assurance not only of liveness, but also of real-time characteristics of system recovery. To assess the feasibility of our theory in practice, we have presented an example based on a prototypical control system, interpreted our theory in terms of its specific requirements, and shown how the theoretical properties hold over an instantiation through type mechanisms.

## Acknowledgments

We thank Xiang Yin and Dean Bushey for their assistance with our example. This work was sponsored, in part, by NASA under grant number NAG1-02103.

## References

[1] ARINC Inc. “Avionics Application Software Standard

Interface.” ARINC Spec. 653, Baltimore, MD, 1997.

- [2] Garlan, D., S. Cheng, and B. Schmerl. “Increasing System Dependability through Architecture-based Self-repair.” *Architecting Dependable Systems*, R. de Lemos, C. Gacek, A. Romanovsky (Eds), Springer-Verlag, 2003.
- [3] Jahanian, F., and A.K. Mok. “Safety Analysis of Timing Properties in Real-Time Systems.” *IEEE Trans. on Software Engineering*, 12(9):890-904.
- [4] Knight, J. C., E. A. Strunk and K. J. Sullivan. “Towards a Rigorous Definition of Information System Survivability.” Proc. 3rd DARPA Information Survivability Conf. and Exposition, Washington, D.C., April 2003.
- [5] Kopetz, H., and G. Bauer, “The Time-Triggered Architecture.” *Proc. IEEE*, 91(1):112-126, Jan. 2003.
- [6] Porcarelli, S., M. Castaldi, F. Di Giandomenico, A. Bonnavalli, and P. Inverardi. “A framework for reconfiguration-based fault-tolerance in distributed systems.” *Architecting Dependable Systems II*, R. De Lemos, C. Gacek, and A. Romanovsky (Eds), Springer-Verlag, 2004.
- [7] Sha, L. “Using Simplicity to Control Complexity.” *IEEE Software* 18(4):20-28.
- [8] Schlichting, R. D., and F. B. Schneider. “Fail-stop processors: An approach to designing fault-tolerant computing systems.” *ACM Trans. Computing Systems* 1(3):222-238.
- [9] Shelton, C., and P. Koopman. “Improving System Dependability with Functional Alternatives.” Proc. 2004 Int’l Conf. Dependable Systems and Networks, Florence, Italy, June 2004.
- [10] Strunk, E. A., and J. C. Knight. “Assured Reconfiguration of Embedded Real-Time Software.” Proc. Int’l Conf. Dependable Systems and Networks, Florence, Italy, June 2004.
- [11] Strunk, E. A., J. C. Knight, and M. A. Aiello. “Distributed Reconfigurable Avionics Architectures.” Proc. 23rd Digital Avionics Systems Conference, Salt Lake City, UT, Oct. 2004.
- [12] Yeh, Y. C. “Triple-Triple Redundant 777 Primary Flight Computer.” Proc. 1996 IEEE Aerospace Applications Conference, vol. 1, New York, N.Y., February 1996.