Dependability Through Assured Reconfiguration in Embedded System Software

Elisabeth A. Strunk John C. Knight

Department of Computer Science
University of Virginia
151 Engineer's Way, Charlottesville, VA 22904-4740
{*strunk | knight*}*@cs.virginia.edu*
+1 434.982.2216 - voice +1 434.982.2214 - FAX

Abstract

In many software systems, properties necessary for dependable operation are only a small subset of all desirable system properties. Assuring properties over the simpler subset can provide assurance of critical properties over the entire system. This work provides a method for constructing systems to be dependably reconfigurable. A system's primary function can have less demanding dependability requirements than the overall system because the system can reconfigure to some simpler function. Reconfiguration thus controls the effective complexity of the system without forcing that system to sacrifice desired, but unassurable, capabilities.

Focusing a system's dependability argument on reconfiguration means that reconfiguration must proceed correctly with very high assurance. The system construction approach in this work also provides a method through which system dependability properties can be shown. To illustrate the ideas in this work, we have built part of a hypothetical avionics system that is typical of what might be found on an unmanned aerial vehicle.

Keywords

Reconfiguration, embedded systems, real-time systems, formal methods, concurrent systems.

1. Introduction

Ultradependable systems have made extensive use of software for some time, and they have a very good overall dependability record. The size and complexity of these systems is increasing, however, and while software development technology is advancing, it is unclear that the pace is rapid enough to match the increase in complexity. While construction of systems of the quality produced in the past might be feasible, the engineering foundation on which the dependability record of those systems rests is weak. The lack of a rigorous dependability argument suggests that the system dependability levels currently achieved are due, at least in part, to the care taken by experienced developers during the design of the software. Careful, but informal, development and review are likely to become less effective as the complexity of the developed software grows beyond the limits of straightforward human comprehension.

In this paper, we present a method for constructing systems whose central design paradigm is assured reconfiguration, in which assurance is achieved by proof. A system with reconfiguration at the center of its assurance argument can allow its primary function to fail and then reconfigure to some simpler function, mitigating any unacceptable failure consequences. Reconfiguration thus controls the effective complexity of the system without forcing that system to sacrifice desired, but unassurable, capabilities.

Current software development practices do provide assurance arguments. While it is unclear that such arguments are convincing for ultradependable software, it is reasonable to assume that engineers can continue to build systems of current complexity with similarly low failure rates. Assurance of complex systems thus reduces to showing that additional functionality will not make future systems less dependable than past ones.

The idea of providing backup systems in case a primary system fails has been employed before in various contexts [5]. Design practices often take into account differences in criticality and exploit them, particularly in the form of simpler software backups or electromechanical back-ups. Examples include the Simplex architecture described by Sha et al. [25] and the Boeing 777 flight control system [31].

Previous research has begun with an existing system and added the capability to reconfigure to the system's original architecture. Our work assumes that construction of complex non-reconfigurable systems which must meet very high levels of assurance cannot be achieved routinely. Thus, reconfiguration should be the driving principle of the system's architecture, and the driving principle of the system's assurance argument. This makes dependable systems both easier to build, since there is no need for extensive replication in functionality that is allowed to fail; and easier to assure, since the reconfiguration architecture is explicitly designed to support assurance.

This work differs from previous work on reconfigurable systems because it presents a comprehensive approach to *assured* reconfiguration. Previous research and the techniques employed in operational systems either have not addressed the issue of assurance of critical reconfiguration properties, or they have been developed in an ad-hoc manner to meet the needs of specific systems. While some systems lend themselves to elegant application-specific solutions, this is not the general case, and so a general mechanism is needed to enable reconfiguration to be a general solution. Furthermore, even when an application-specific solution is available, designers can benefit from the improved support for verification and validation provided by a more general approach.

This work sets forth an approach to system construction that ensures dependability properties by ensuring critical functional properties and critical reconfiguration properties. The approach accomplishes this by: (1) introducing a formal definition of reconfiguration and an associated set of high-level, general properties; (2) constructing an architecture that guarantees the high-level reconfiguration properties; and (3) making non-crucial functionality fail-stop [23], so that a failure in one application does not cause a failure in others. Showing that a specific system complies with the architecture's properties provides assurance of reconfiguration for that system.

This paper is organized as follows. Section 2 elaborates the advantages of reconfiguration in dependable systems. Section 3 discusses related work. Section 4 presents an informal discussion of reconfiguration, and Section 5 describes a candidate architecture for implementing reconfiguration. Section 6 presents an argument for reconfiguration assurance. Section 7 presents an example avionics system that instantiates the architecture. Section 8 concludes the work.

2. Reconfiguration as an Architectural Driver

Software has many advantages, such as flexibility and economy, that make its use tempting to application designers. Adding automation to safety-critical embedded systems can sometimes increase the safety of the overall system: embedded systems can often assist or replace human operators, who are naturally error-prone [21]. Such automation can effectively make the system more dangerous, however, because of a consequent reduction in the margin for error that would normally be included. Furthermore, as the software functionality becomes more extensive and complex, its overall function and safety implications become much more difficult for system designers to comprehend. Lack of comprehension introduces opportunities for error, and in systems that must be dependable, those errors could well have unacceptable consequences.

Managing complexity by limiting the functionality included in a system is a poor choice economically and thus is likely to be ignored in the business world. The most practical solution is to make the task of program comprehension manageable. While this strategy does not solve the problem, it helps the humans who build and analyze ultradependable software understand their systems well enough to determine whether the systems satisfy their dependability criteria.

2.1 Bounding the dependability problem

In many cases, much of the functionality included in a system is not essential for system safety. For example, the autopilot system on a commercial aircraft could contribute to an accident, but cessation of its function is unlikely to have catastrophic consequences as long as the pilot is informed. Such a system may not need to be *ultradependable* but, rather, *fail-stop* [23]: the autopilot must either work correctly or stop and alert the pilot. The complete avionics system needs to be able to operate without the autopilot so that safety is not compromised if the autopilot fails. In some cases, the remaining subsystems will alter their modes of operation to compensate for the failed subsystem. In this way, system reconfiguration can compensate for subsystem faults.

Dependability is often achieved through fault masking, but masking becomes increasingly difficult with rising complexity. Reconfiguration can reduce the effective complexity of critical function and limit the amount of software that is crucial to dependable operation.

2.2 Reconfiguration for embedded systems

Building reconfiguration into the basic design of a system allows the distinction between crucial and noncrucial function to be made clearly at the requirements level and set forth in the system's specification. Thus, designers can specify the properties they want, rather than employing individual mechanisms that work towards those properties (e.g., fault-tolerant components).

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. Our reconfiguration framework is unique (to the best of our knowledge) in that it is designed explicitly to facilitate formal system analysis based on a system's reconfiguration capability.

2.3 Reducing resource requirements

Achieving hardware dependability through pure replication is currently nontrivial and will be increasingly difficult in the future because hardware component failures become more common as the number of components increases. Although lower component cost means that components can be more easily replicated, hardware replication—and the environmental shielding, power, and cooling facilities that must accompany it—adds weight and takes up space, leading to higher operational costs, and might be impractical in many circumstances.

In a system where hardware faults are masked, there has to be sufficient equipment available to provide full service if the anticipated number of component failures occurs during the maximum time planned for continuous operation. The total number of required components is thus the sum of the maximum number expected to fail during the longest planned mission and the number needed to provide full service. Though possible, loss of the maximum number expected to fail in a system with significant replication 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.

Reconfiguration offers a trade-off between functionality and hardware resources. This tradeoff is exploited by building system hardware with less provision for coping with hardware faults than normally might be included. In a system with a reconfigurable architecture, the total number of hardware components can be as low as the sum of the maximum number expected to

fail during the longest planned mission and the *minimum* number needed to provide the most basic acceptable service. If the system were designed so that this number equals the number of components needed to provide full service, then during routine operation (the vast majority of the time), the system would operate with no excess equipment. This saves power, weight and space.

2.4 Replacing Ad-Hoc Solutions

For many years, reconfiguration has been an effective solution to problems of complexity and uncertainty in a variety of systems. These systems' reconfiguration mechanisms generally rely on properties of the applications that make them amenable to reconfiguration. While our architecture was not created to benefit these systems specifically, in many cases it can still contribute to their development because of the verification potential it provides. We believe it will be more cost-effective to use our proofs than to create high-level proofs for specific systems, and the proofs can contribute to assurance in a way that complements system testing.

3. Related work

3.1 Other work in reconfiguration

Other researchers have used reconfiguration to increase system dependability in a variety of contexts. Graceful degradation, for instance, provides fine-grained control of functionality during degradation. It could be accomplished by reconfiguring to some other operational specification when a failure occurs. Generally, however, our reconfiguration architecture advocates more coarse-grained use of remaining components, sacrificing some of graceful degradation's postulated utility in order to permit rigorous and practical analysis of reconfigurable software. Quality of service is similar to graceful degradation but is generally used to refer to specific performance aspects of a system, e.g., video quality. Our architecture permits reconfiguration to impact system function much more broadly, changing the function more dramatically or replacing it altogether.

Shelton and Koopman have studied the identification and application of useful alternative functionalities that a system might provide in the event of hardware component failures [26]. Their work is focused on reconfiguration requirements and can therefore be used in the design of reconfigurable systems, but does not address assurance explicitly. Sha has studied the implemen-

tation of reconfiguration in fault tolerance for control systems [24], although his work does not focus on assurance. Likewise, Garlan et al. [9] have proposed the use of software architectural styles as a general method of error detection, and reconfiguration 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 in a given operating environment [11]. For networked systems, the loss of a moderate number of randomly-distributed components must be expected. Reconfiguration is employed only in the case of large numbers of failures, or if a common cause is likely. The main challenge of networked system reconfiguration is managing system scale.

Our work is part of a framework for using reconfiguration in embedded real-time systems. Embedded system reconfiguration requirements are similar to networked system requirements, with three key differences: (1) embedded systems are much smaller and thus can be tightly controlled; (2) they often 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, leading to assurance requirements of a functional transition that are much more demanding.

The notion of reconfiguration to deal with faults has also been used extensively in production safety-critical and mission-critical embedded systems. For example, the Boeing 777's primary flight computer contains two sets of control laws: the primary control laws of the 777 and a set of simpler, less efficient control laws as a backup [24]. It also has a mechanical backup. The Airbus A330 and A340 employ a similar strategy [27], as have embedded systems in other critical domains. Existing approaches to reconfigurable architectures are, however, ad hoc; although the system goals are achieved, the result is inflexible and not easily assured.

Finally, reconfiguration has been used in the construction of software systems where the system must be dependable but reconfiguration does not support this explicitly. Such systems include spacecraft, where the main goal of the mission can only be carried out once the craft has reached its destination, and reaching that destination is a challenge in itself [16]. A significant body of

work exists in analysis of space system hardware component reconfiguration, but little research in software aspects of their reconfiguration has been completed (a notable exception in the literature is the Corot mission software [8]). Reconfiguration also appears in "intelligent" control systems (such as described by Bateman et al. [4]), and in adaptive reconfigurable computing [17].

3.2 Fail-stop software

Halting a system to allow it to reconfigure is unlikely to pose challenges that are significantly more difficult than those posed by construction of the system's function, if that system is operating normally. If part of the system has failed, however, its failure semantics must be guaranteed to support the needs of the reconfiguration structure. Work on assuring failure semantics of software systems is discussed in this section.

Schlichting and Schneider documented the concept of a fail-stop computer as a building block for safety-critical systems, and they introduced a programming approach based on *fault-tolerant actions* (FTAs) in which software design takes advantage of the computer's fail-stop semantics [23].[1] They define a fail-stop computer to consist of one or more processing units, volatile storage, and stable storage. The failure semantics of a fail-stop computer are [23]:

FS1: It stops executing.

FS2: The internal state and contents of the volatile storage connected to it are lost.

Stable storage must not be affected by a failure.

An embedded system of the type assumed in this work is made up of a collection of fail-stop computers. If one computer fails, the others poll its stable storage to determine the state it was in when it failed.

The software analog of fail-stop machines is the concept of *safe programming* introduced by Anderson and Witty [2]. Safe programming requires (in part) modification of the postcondition for a program by adding an additional clause that allows the program to terminate without modi-

1. Schlichting and Schneider's work discusses fail-stop *processors*; we use the term *fail-stop computer* instead, to represent the set of physical components required to provide the semantics they describe.

fying the state and signal a failure. A *safe* program in this sense is one that satisfies its (modified) postcondition. The problem of assurance has thus been reduced to one of assuring comprehensive checking of the program's actions—a much simpler task than assuring overall functionality.

Related to safe programming is the idea of a *safety kernel*. The idea has been studied by several researchers. Leveson et al. use this term to describe a system structure where safety-relevant functionality is gathered in a centralized location [14]. Rushby has defined the term more generally as a small component that guarantees the enforcement of a specific class of properties [22]. Wika and Knight characterize classes of safety policies that might be enforced [30]. Burns and Wellings define a safety kernel as a *safe nucleus* and a collection of *safety services* [6].

Knight has introduced the term *protection shell* to describe a policy enforcement mechanism that checks program outputs rather than inputs to particular function calls. Protection shells guarantee software component properties by checking that properties of those components' outputs hold. Shell analysis is much simpler than system analysis because the shell for each component is (1) simpler than the component itself and (2) designed explicitly to facilitate analysis [30].

3.3 The PVS proof system

We used the Prototype Verification System (PVS) to construct and check the formalisms and proofs in this work, so we include a brief introduction here. We will refer to the concepts presented here in Sections 5 and 6.

3.3.1. The PVS language. The PVS language is a higher-order logic based on type theory. Subtypes are defined by adding a predicate to a supertype. A subtype `Sub` of supertype `Super` with property `P` could be defined as `{s: Super | P(s)}`. `P` would then have to hold over any instance of `Sub`. In functions, if the function takes a parameter `s` of type `Sub`, then it can assume `P(s)` holds. In some cases, the consistency of a specification's type system is undecidable; this leads to the generation of *type-correctness conditions* (TCCs), theorems that must be proven in order for the system to be considered type-correct.

PVS is a functional language that provides a mechanism to construct record types. Each element of the record must have a declared type, and any element of any instance of the record must

be a member of the corresponding type. Again, if type predicates of the instance are undecidable, TCCs will be generated.

3.3.2. The PVS system. The PVS system allows a developer to create specifications, state theorems over those specifications, write proof scripts for the theorems, and then mechanically check the scripts to see whether they do indeed prove the properties. Proof scripts consist of a series of LISP-like commands that mechanically manipulate proof sequents. A valid proof script is a sequence of commands that will, when applied to a putative theorem, transform that theorem to "`true`". The theorem starts out as the only statement in the proof consequent, so that the proof of a theorem `T` is a proof that "`true` $\Rightarrow$ `T`." Intermediate lemmas and other theorems, as well as axioms (which require no proof themselves) can be imported into a proof, and are listed as statements in the proof's antecedent. Type predicates can also be imported into a proof antecedent.

4. An informal model of reconfiguration

If a system's dependability argument is centered on its reconfiguration properties, those properties must be stated clearly and must meet very strict dependability criteria. This section describes the intuitive properties that a reconfigurable system must exhibit, and an informal discussion of an architecture that guarantees those properties. Section 5 formalizes these concepts.

4.1 Reconfigurable fail-stop systems

Fault-tolerant actions (FTAs), introduced by Schlichting and Schneider, are a building block for programming systems of fail-stop computers. Briefly, an FTA is a software operation that either: (1) completes a correctly-executed action $\mathcal{A}$ on a functioning computer; or (2) experiences a hardware failure that precludes the completion of $\mathcal{A}$ and, when restarted on another computer, completes a specified recovery protocol $\mathcal{R}$. 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 to construct application software to mask the effects of a failure and how to construct proofs that state is properly maintained.

In the original framework, an FTA's recovery protocol may complete only the original action, either by restarting the action or by some alternative means. Our framework takes a broader view

of the recovery protocol, where $\mathcal{R}$ might be the reconfiguration of the system so that the next $\mathcal{A}$ will complete some useful, but often *different*, function. An FTA in this framework, then, requires that the system either carries out the function requested, or puts itself into a state where the next action can execute some appropriate function (although possibly not the one requested).

4.2 Informal definition and applicability conditions

Our extensions to Schlichting and Schneider's framework are driven by the needs of reconfigurable systems. Informally, we define reconfiguration to be *the process through which a system halts operation under its current source configuration* c_i *and begins operation under a different target configuration* c_j. This is a very broad definition, and could mean many different things in classes of systems with broadly varying requirements. In order to refine this definition, we consider the class of systems with the following properties:

- The system is made up of a set of periodic applications $A = \{a_1, a_2, ..., a_m\}$. Each a_i in A possesses a set of possible functional specifications $S_i = \{s_{i1}, s_{i2}, ..., s_{in}\}$ and always operates in accordance with one of those specifications unless engaged in reconfiguration. Any functional dependencies among the applications in A must be acyclic.
- Applications are synchronized so that the execution of all applications is periodic at the system level. Each application may execute one or more times during the system period. Synchronization may occur over a sparse time base [12].
- Worst-case execution times, including worst-case time to initialize data in a new configuration, can be determined for each function in a specification.
- It is possible to know in advance all of the desired potential system configurations $C = \{c_1, c_2, ..., c_p\}$ and how to choose among 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 the dependability goals of the system.
- The software is running on fail-stop or fail-silent computers, as in the Delta-4 project [20], and commits to stable storage are atomic, as in the Fault Tolerant Multiprocessor [15].

These requirements describe most critical embedded systems. They are, in a sense, a relaxation of the requirements on any system operating in real time. The periodicity and execution time requirements are required for any real-time system, and in traditional systems only one specification and one configuration are available to the system.

- System function can be restricted during system reconfiguration. Time bounds on function restriction are discussed in section 4.4.
- There is a reconfiguration trigger; the source of the trigger might be a hardware failure, a software functional failure, a failure of software to meet its timing constraints, or a change in the external environment that involves no failure at all. We neither provide nor assume any specific error detection mechanisms.

The above properties are those required for our work to apply to a system in general, and they are assumed in our architecture. In our specific formal model, we also assume:

- All applications operate with the same period.
- The system period is equal to the application period.
- 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 initialization of data such as control system gains).
- Each application commits its results to stable storage at the end of each real-time frame.
- Any dependencies between applications require only that the independent application be halted before the dependent application computes its transition condition.

These assumptions limit the class of systems to which the specific formal model we have built will apply. We added them to create a manageable starting point for the proofs we constructed—not only are they intellectually difficult to manage, the mechanical *checking* of the proofs (described below) presented here on the fastest machines available to the authors takes approximately 24 hours. We plan to relax them in future work, as part of an overall program to take the starting point presented here and make it more accessible to industrial developers.

In our formal model, we assume that an application is composed of a set of modules, and application properties of interest can be constructed through conjunction of module properties of interest. Similarly, system properties of interest can be constructed through conjunction of application properties of interest. This assumption could easily be relaxed through a small change in the formal model that would actually simplify the model. We included it here to show how existing work on error detection could be incorporated into the formal architecture; it could be removed in a system where it was not needed.

Using these assumptions to refine the informal notion, reconfiguration between two configurations c_i and c_j is the process R for which:

- R begins at the same time the system is no longer operating under c_i;
- R ends at the same time the system meets the precondition for c_j;
- c_j is the proper choice for the target specification at some point during R;
- R takes no more than the maximum time for which the application can be nonoperational; and
- a transition invariant holds during R (disallowing random state changes during R).

These conditions will be formalized below.

4.3 Reconfigurable fault-tolerant actions

4.3.1. Fault-tolerant actions. The timing model we use to describe system reconfiguration is an expanded version of the timing model Schlichting and Schneider used in their characterization of fail-stop computing. In the approach presented here, the basic software building block is a *reconfigurable application*, referred to as an application in the remainder of this work. An application has a predetermined set of specifications with which it can comply (as described above) 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 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 actions in other applications. These issues did not arise in

the previous formulation of FTAs since failures were masked. Adding the possibility of reconfiguration necessitates a distinction between *application* FTAs (AFTAs) and *system* FTAs (SFTAs).

4.3.2. Application Fault Tolerant Actions. An AFTA is an action encompassing a single unit of work for one application. This maps to one standard execution cycle in a periodic application that is functioning normally. If the application is reconfiguring, an AFTA consists of the execution cycle in which the reconfiguration is signaled and the sequence of steps carried out to effect the reconfiguration. This sequence is:

- An error is detected or some other reconfiguration signal is generated.
- The application postcondition is met. The postcondition for an application is defined to be the precondition for a transition to a new configuration.
- A new configuration is selected.
- The application state is modified so that the new configuration can be put into effect.
- The application is set to operate under the new configuration (may occur automatically).
- State (e.g., control loop gains) is initialized to be consistent with the new specification.

4.3.3. System Fault Tolerant Actions. 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 a reconfiguration can affect several applications, a mechanism to determine which application reconfigurations are necessary to complete an SFTA is required. We introduce the System Control Reconfiguration Analysis and Management (SCRAM) kernel to implement the *external* reconfiguration [19] portion of the architecture. Possible configurations to which the system might move to complete its SFTA are defined by a statically-determined set of valid system transitions. A function to determine which transitions must be taken under the different operating

circumstances that might arise is included. The SCRAM kernel will: (1) accept reconfiguration signals; (2) determine the configuration to which the system should move; and (3) send appropriate signals to the individual applications to cause them to reconfigure, if needed.

Our approach differs slightly from the approach taken by Schlichting and Schneider. In their approach, if a failure occurs, the recovery for the FTA is executed after the failure. The recovery for an SFTA during which a reconfiguration signal is generated consists of three parts:

1. Each executing AFTA establishes a predetermined postcondition so that the corresponding application can reconfigure, if necessary.
2. Each application that must reconfigure establishes the condition to transition to operation under the new state, and all other applications execute an additional standard AFTA.
3. Each application that must reconfigure establishes its precondition, meaning that all state associated with the AFTA has been initialized and the application is functioning normally, and all other applications execute an additional standard AFTA.

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

4.4 Real-time guarantees on fault-tolerant actions

Schlichting and Schneider's work includes a discussion of system temporal properties in the event of multiple successive failures, including a method to guarantee liveness. We extend their framework to cover the timing elements of reconfiguration for systems that must operate in hard real time. Depending on system requirements, any reconfiguration signals that are generated while a system reconfiguration is in progress can be either addressed immediately or buffered until the next stable storage commit of all applications. In the worst case, failures 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 sequence of transitions to some safe system configuration C_s. In other words, for the long-

est configuration sequence C_1, C_2, ..., C_s, the maximum restriction time is $\Sigma_{i=1 \text{ to } s-1} T_{i, i+1}$, 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 maximum time to successful action completion over all possible system transitions would be $\max\{T_{i, s}\}$.

Cyclic reconfiguration is possible due to repeated failure and repair or rapidly-changing environmental conditions, and in this case the time between two successful actions could be infinite. Potential cycles can be found through a static analysis of permissible transitions. They can be dealt with by forcing a check that the system has been functional for a minimum period of time (in a safe or universally appropriate configuration) before a subsequent reconfiguration takes place.

5. Candidate reconfiguration architecture

This section describes one possible architecture that facilitates the refinement of the properties listed above into a set of properties that can be shown of an individual system. If a system is built using this architecture and shows the properties required of the architectural elements, the developer will know that the properties that assure reconfiguration have been met. We have formally modeled the architecture in PVS [29], although for brevity, our discussion here is mostly informal. The proofs that the architecture implies the properties are discussed in Section 6.

5.1 Application architecture

In the candidate architecture, a system is composed of a set of applications of the sort characterized in Section 4.2. Each application implements a set of specifications and provides an interface for *internal* reconfiguration [19].

5.1.1. State. To guarantee properties over the architecture's state, the state is represented explicitly as a set of mappings from data identifiers to data values. In a fail-stop computer, these data elements would be kept in the system's stable storage.

5.1.2. Modules. The internal structure of an application is depicted in Figure 1. Each application is built of a set of modules. A module represents some set of functions that operate over a particular set of data elements and guarantee certain properties over those data elements. Each module includes predicates defining the invariant, precondition, transition condition, and postcondition for that module's state. Each function in a module interface presents a set of functional *service levels*. Functions within a module carry out different computations based on the module's service level, and the guarantees they provide are also service level-specific.

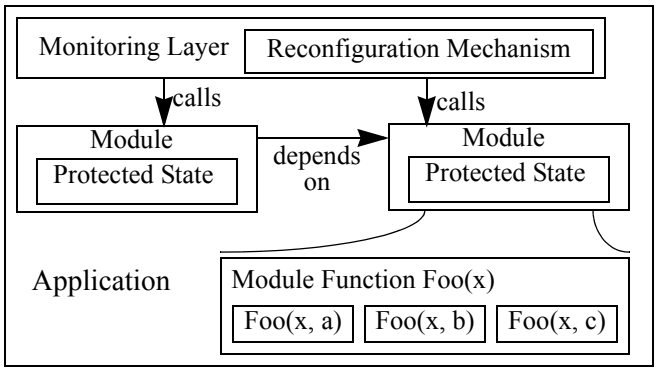

Figure 1. Application Structure

5.1.3. Application reconfiguration. Application configurations are constructed from combinations of module services. Configuring the modules to provide the correct services for a specific application configuration can be done in at least two ways. First, the application can contain a mapping from each application configuration to the module service level appropriate for that configuration. The top-level execution loop would take the configuration mapping from modules to appropriate service levels as a parameter. Alternatively, each module's state could include a module-specific *service level parameter*. In this case, a call to `foo(x)` would effectively become a function call to `foo(x, svclvl_parm)`, where `foo(x, svclvl_parm)` is the proper version of `foo(x)` for the module configuration corresponding to `svclvl_parm`. Our formal model provides for either configuration method. The example in Section 7 uses the function mapping to choose reconfiguration functions, and the service level parameter to choose application functionality.

Properties of the application are decomposed into properties of the modules, so that the conjunction of module properties (e.g., postconditions) gives the corresponding application property. Dividing properties in this way creates a clear interface in the architecture for the use of existing work on guaranteeing software properties (e.g., safe programming, safety kernels, and protection

shells, as described in Section 3.2). Application-level errors that span several modules could be detected by creating a protection shell that encompasses the individual modules.

The functions that are called to effect reconfiguration are defined at the application level to change modules' internal state in a coordinated way. Each application contains operations specific to each service level that carry out the following functions:

- Normal execution
- Execution in preparation for a possible reconfiguration
- Halting when a reconfiguration has been signaled
- Preparing to transition to a new configuration

We refer to these functions collectively as the application's *monitoring layer*, the architectural component responsible for (1) overall supervision and control of application function, and (2) coordination of AFTAs in the context of their SFTAs.

Any reconfiguration signal that is generated or recognized by an application during computation of a module function causes control to be returned to the monitoring layer. The monitoring layer then relays the signal to the SCRAM and reconfigures the application as directed by the SCRAM. At the beginning of each non-reconfiguration execution cycle, the monitoring layer checks for any reconfiguration signals that have been sent to the application from the SCRAM during the previous cycle. If it sees that one has been sent, it executes a modified version of the functionality for the current configuration that saves any state that might be necessary for a possible upcoming reconfiguration to take place. In the following cycle, the monitoring layer would then compare the old and new configurations; if the two are the same, it simply switches back over to normal execution, and if the two are different, then reconfiguration proceeds.

5.1.4. System reconfiguration. Each period, the SCRAM checks to see whether some application is waiting to be told its new configuration. Since reconfiguration proceeds in lock-step, this means that they are all waiting. If no other signal has been generated (a signal occurring at this stage is the only case where the applications' reconfiguration status might not be synchronized), the SCRAM tells each application its new configuration. The delay from signal generation to new

configuration updates allows an implementation time to choose the new configuration. After the new configuration is assigned, the applications will prepare for a transition if one is needed.

If no application is waiting to be told its new configuration, then the reconfiguration status of each application is updated to reflect the last cycle's completed computation. The applications then carry out their execution for the current cycle.

5.2 Reconfiguration Specification

Domain experts will define system *configurations* that provide acceptable system-level services. System reconfiguration is the transition from one configuration to another. The details of application interaction at the system level are captured in the *reconfiguration specification*. The reconfiguration specification includes:

- *Operating environment.* Which configuration is most useful to the user at the point when reconfiguration is required might depend on many factors; examples include aspects of the operating state, time of day, and stage of flight. Also, the failure status of some system components can be modeled as part of the environment since that status is given rather than effected.
- *System configurations.* The reconfiguration specification includes information on dependencies among applications, overall system configurations, system transitions, and how to choose a transition appropriate to circumstances that hold when a reconfiguration is signaled.
- *System transitions.* The type representing a system transition is parameterized over the set of possible system configurations and the set of possible environmental states for the system. It contains the source and target configuration of a system transition, and the environmental states under which that transition would be appropriate. The reconfiguration specification includes a function to choose among potential transitions.

5.3 State traces and the SCRAM

The PVS formalization of the architecture sets out time as an explicit element of state, and temporal predicates are written as predicates that restrict the value of time in conjunction with restrictions on the value of state. In order to specify state-change restrictions over time, the model

Table 1: Reconfiguration Stages

Frame	Stage	Action	Predicate
1 (start)	Application *i*: `interrupted` All other applications: `normal`	None	None
2	Application *i*: `halting` All other applications: `exec_halting`	Applications execute functions that anticipate reconfiguration	Application postconditions
3	SCRAM: `prepare(`C_t`)` → `all apps`	Applications prepare to transition to C_t	Application transition conditions for C_t
4 (end)	SCRAM: initialize → all apps	Applications initialize, establish operating state for C_t	Application preconditions for C_t

defines possible system traces that can be legally generated by a system which complies with a particular reconfiguration specification.

The assumptions on synchrony and equal execution times mean that reconfiguration either: (1) takes 4 cycles to complete successfully, including the cycle during which the reconfiguration signal is generated; (2) takes 2 cycles to complete successfully because no change is needed (an unlikely but possible case); or (3) takes less than 4 cycles because a second signal is generated and serviced before the first reconfiguration is complete. In the latter case, the first reconfiguration is simply ignored. The different stages of reconfiguration are shown in detail in Table 1. In the table, *i* denotes the application that generates a reconfiguration signal, all applications will reconfigure, and the new system configuration will be C_t.

System reconfiguration is effected by the SCRAM kernel, which ensures that state traces happen in an appropriate sequence. The SCRAM has a standard interface so that applications can be easily added to or removed from the system from a reconfiguration point of view. The standard interface enables the SCRAM to be reused across many different systems. The reconfiguration specification contains all of the system-specific information that must be provided to the SCRAM to ensure that appropriate reconfigurations occur when necessary.

5.3.1. Application execution. Application execution is modeled by specifying the actions of the monitoring layer for an application. The monitoring layer chooses each cycle whether to carry out an application's normal function, or to execute one of the reconfiguration stages for the application. Its choice is determined by the system reconfiguration state value for that application.

Failures are modeled as an undefined function that can either return the same reconfiguration state or indicate that a reconfiguration signal has been generated. Failures can be introduced at any time when the system is not in a safe configuration. Concurrent execution of all system applications must leave the system in a state equivalent to that which would result from executing the applications in a sequence that does not violate their dependencies.

5.3.2. State trace. A reconfigurable system has a number of elements of system state that do not belong to any application. These elements include the reconfiguration state of the system and the time that the system state is true of the system. All elements of system state are collected in the `sys_state` type. State traces are sequences of instances of this type. We have created a set of functions that define rules for state traces that satisfy the specification constraints. Essentially, the state trace functions will map to the SCRAM's execution, with the help of bus characteristics and processor clock synchronization.

A valid sequence of system states is one where: (1) the beginning state is a non-reconfiguration state; (2) the system always eventually reaches a non-reconfiguration state; (3) any state is equal to the previous state updated by the SCRAM and then updated by the sequence of application executions; and (4) the system state is synchronized with the environment. The `sys_trace` type formalizes these requirements. We use it in Section 6.2 to express reconfiguration properties.

The temporal structure in our model allows additional reconfiguration signals to be generated during a reconfiguration. In this case, the reconfiguration is simply restarted. If a new specification has been chosen, the reconfiguration is completed with the new specification's conditions as its starting conditions; otherwise, the old specification's conditions are used. Also, the SCRAM's synchronization mechanism can easily be extended as needed to support richer dependencies among applications, as long as those dependencies are acyclic and enough time is available. Given a specification of dependencies, the SCRAM could preserve the dependencies by checking each cycle to see if the independent application has completed its current configuration phase. Only if that phase were complete would the SCRAM signal the dependent application to begin its next stage. Unnecessary dependencies permitted here could also be relaxed, and thus time to

reconfigure shortened, by removing any unnecessary intermediate stages or allowing the applications to complete multiple sequential stages without signals from the SCRAM.

5.4 Example implementation platform

To illustrate a potential use of the architecture, we briefly describe a possible implementation platform, illustrated in Figure 2. The platform includes a set of processing elements that communicate via an ultra-dependable, real-time data bus. Each processing element consists of a fail-stop computer that executes an ultradependable real-time operating system. An example fail-stop computer might be a self-checking pair; an example data bus might be one based on the time-triggered architecture [13]; and an example operating system might be one that complies with the ARINC 653 specification [3]. Sensors and actuators are connected to the bus via interface units that employ the bus protocol.

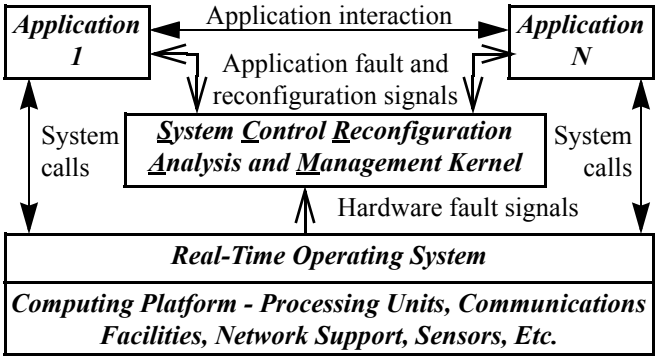

Figure 2. Logical System Architecture

Each application operates as an independent process mapped to some processing element. Applications communicate by sharing state through stable storage. The SCRAM executes on a fail-stop computer, and its functionality is implemented as a set of fault-tolerant actions in the original sense of Schlichting and Schneider so that any failures are masked.[1] It communicates with applications through variables in stable storage. When reconfiguration is necessary, it sets the *configuration_status* variable of each application to a sequence of values on three successive real-time frames. The three values are *halt*, *prepare*, and *initialize*, reflecting the AFTA stages described in Section 4.3.2. At the beginning of each real-time frame, each AFTA reads its *configuration_status* variable and completes the required action during that frame.

1. Note that this means the worst-case time to transition must be added to the worst expected time for the SCRAM to complete its FTA.

The above is only one possible platform for implementing the architecture. The architecture specification is a set of application characteristics and interactions. An implementation of a system that has the architecture need only exhibit the properties set out in the architecture specification. The specifics of an argument that the system exhibits those characteristics will vary widely across different implementations. A system where all applications run on a single processor, for instance, need not address network communication reliability; and if the applications are written in Ada, then the Ada runtime executive can be used and so no operating system is needed. With a mechanism to ensure atomicity of stable storage commits and a mechanism to verify the Ada code against the PVS system specification, the assurance argument would be complete.

6. Proof Structure

We have created a reconfiguration assurance argument that 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 a definition of system reconfiguration; and (3) proofs of the theorems. These proofs constitute a proof that the architecture satisfies the definition. With this verification framework in place, any instance of that architecture will be a reconfigurable system with the stated formal properties, as long as the TCCs of the formal model's type system can be proven.

The reconfiguration architecture is specified in PVS, proofs of the putative theorems have been constructed, and the proofs have been checked 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 the formal architecture. The result is an assurance argument based on proof. Figure 3 depicts the relationship among formal elements discussed here, and how they fit into the broader picture of assured software development.

6.1 Reconfiguration proof structure

We have constructed a set of types in PVS that defines a reconfigurable system specification and architecture. Any specification that instantiates the type system will thus possess the properties of the formal model. Conformance can be checked by writing the system as an instance of the specification record type. If PVS is unable to determine whether the system has the appropriate

type (and thus, has the appropriate properties), it will issue TCCs that the specifier must prove. If the instance does not type-check, it might not have the architecture's high-level properties.

The proofs of high-level properties are proofs over state traces that can be generated, given a specific reconfiguration specification. Properties for individual applications, and the system-specific data that must be input to the SCRAM, are all encoded in the type system. We proved the properties over the combination of: (1) the state trace functions; and (2) the type predicates from the reconfiguration specification and its constituent application specifications.

The property proofs we have created are quite lengthy, since mechanical proofs must often be much more detailed than logical proofs. They have been mechanically checked with the PVS system, thereby ensuring that what is stated has been proven, given that either: (1) PVS does not contain any faults that are activated by the checking of the proofs; or (2) errors that can be generated by faults in PVS do not cause unprovable theorems to be provable. The proof scripts are not included here, but can be found elsewhere [29].

6.2 Reconfiguration definition

We define reconfiguration as a set of required high-level properties. Defining reconfiguration properties in an abstract sense allows us to argue that the general requirements of reconfiguration have been met. The model was constructed and refined to enable proof of these properties.

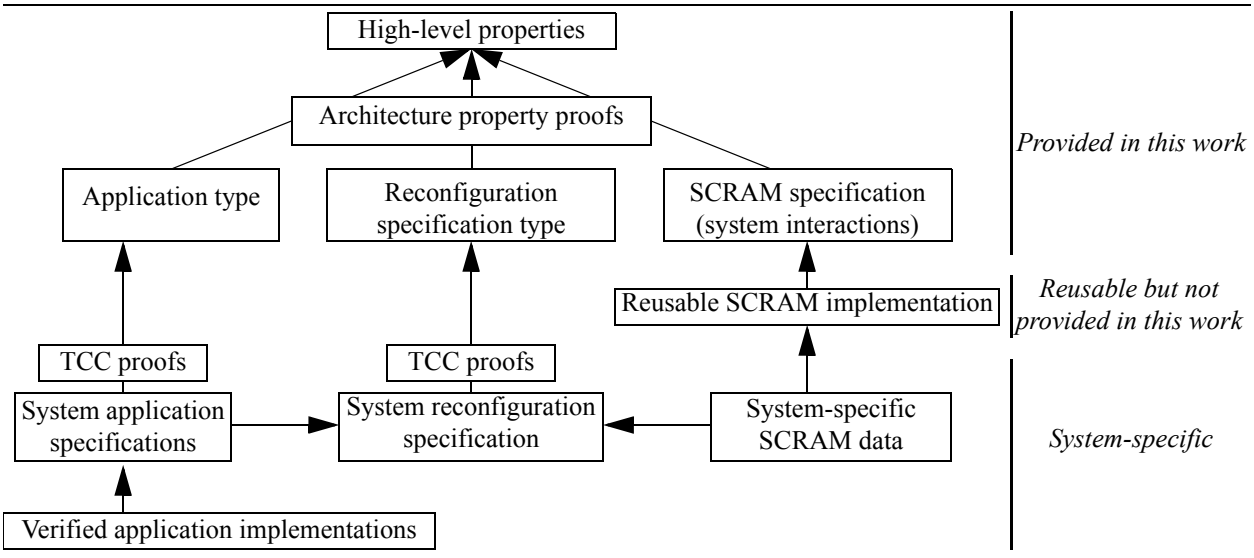

Figure 3. Software Products for a Specific System

We begin by expanding our previous informal description of reconfiguration to:

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

An action thus comprises the correct execution of all applications $a_i \in A$ 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 application fault-tolerant action. If only a_i must reconfigure, then $\forall a_j \neq a_i, f'(a_j) = f(a_j)$.

Formally, we characterize reconfiguration as five properties, shown in Figure 4, that must hold over any sequence of states that is a subsequence of a valid `sys_trace` and whose reconfiguration status is not `normal`. The sequence of states is represented by the reconfiguration type, which consists of two natural numbers that represent the beginning and ending system execution cycle for that reconfiguration. All other state for the reconfiguration is represented in the sequence of system states that make up the reconfiguration. The sequence is bounded at the beginning by a signal generated by some application; and at the end, by either a second signal generated from some application, or by all applications' having finished initialization and returned to `normal` status. This is defined formally in the `get_reconfigs` function.

`CP1` defines what makes up a reconfiguration, and is essentially a repetition of the `get_reconfigs` function, included for clarity in the discussion of abstract properties.

`CP2` states that either: (1) a signal was generated before applications were notified of the new configuration, and so the reconfiguration ends with the system in the same configuration it was in when the reconfiguration began; or (2) the reconfiguration reached the notification stage, and so at the end of the reconfiguration (regardless of whether the reconfiguration were interrupted), the system will be in a new configuration which is consistent with the system configuration and environmental conditions in effect when the new configuration was selected.

```
CP1: THEOREM
   FORALL (s: sys_trace, r: (get_reconfigs(s))) :
      r`start_c < r`end_c AND reconfig_start?(s, r`start_c) AND
      reconfig_end?(s, r`end_c) AND
      FORALL (c: cycle) : (r`start_c < c AND c < r`end_c => NOT reconfig_end?(s, c))
CP2: THEOREM
   FORALL (s: sys_trace, r: (get_reconfigs(s))) :
      (r`end_c - r`start_c = 1 AND s`tr(r`end_c)`svclvl = s`tr(r`start_c)`svclvl) OR
      EXISTS (c: cycle) : r`start_c <= c AND c <= r`end_c AND
         s`tr(r`end_c)`svclvl = s`sp`choose(s`tr(c)`svclvl, s`env(c*cycle_time))
CP3: THEOREM
   FORALL (s: sys_trace, r: (get_reconfigs(s))) :
      (r`end_c - r`start_c + 1)*cycle_time <=
         s`sp`T(s`tr(r`start_c)`svclvl, s`tr(r`end_c)`svclvl)
CP4: THEOREM
   FORALL (s: sys_trace, c: cycle) :
      % The function invariant holds
      inv(s`sp, s`tr(c)`svclvl, s`tr(c)`st) OR
      (c > 0 AND
       FORALL (app: (s`sp`apps)):
         % The application generated a signal during the
         % preparation stage and still meets the last configuration's invariant
         (s`tr(c)`reconf_st(app) = interrupted AND
            (s`tr(c-1)`reconf_st(app) = halting OR
               s`tr(c-1)`reconf_st(app) = exec_halting) AND
            (s`sp`SCRAM_info`configs(s`tr(c-1)`svclvl)(app) /=
               s`sp`SCRAM_info`configs(s`tr(c)`svclvl)(app)) AND
            inv(app`modules, app`svcmap(s`sp`SCRAM_info`configs
               (s`tr(c-1)`svclvl)(app)), s`tr(c)`st)) OR
         % The application did not receive a signal during the
         % preparation stage and meets the new invariant
         inv(app`modules, app`svcmap
            (s`sp`SCRAM_info`configs(s`tr(c)`svclvl)(app)), s`tr(c)`st))
CP5: THEOREM
   FORALL (s: sys_trace, r: (get_reconfigs(s))) :
      % the reconfiguration was not interrupted and some application reconfigured
      (r`end_c - r`start_c = 3 AND FORALL (app: (s`sp`apps)) :
            (s`sp`SCRAM_info`configs(s`tr(r`start_c)`svclvl)(app)
               /= s`sp`SCRAM_info`configs(s`tr(r`end_c)`svclvl)(app) AND
               pre(app`modules, app`svcmap(s`sp`SCRAM_info`configs
                  (s`tr(r`end_c)`svclvl)(app)), s`tr(r`end_c)`st) OR
            (s`sp`SCRAM_info`configs(s`tr(r`start_c)`svclvl)(app)
               = s`sp`SCRAM_info`configs(s`tr(r`end_c)`svclvl)(app) AND
               inv(app`modules, app`svcmap(s`sp`SCRAM_info`configs
                  (s`tr(r`end_c)`svclvl)(app)), s`tr(r`end_c)`st))))OR
      % the reconfiguration was not interrupted but no application reconfigured
      (r`end_c - r`start_c = 2 AND FORALL (app: (s`sp`apps)) :
            (s`sp`SCRAM_info`configs(s`tr(r`start_c)`svclvl)(app)
               = s`sp`SCRAM_info`configs(s`tr(r`end_c)`svclvl)(app) AND
            inv(app`modules, app`svcmap
                  (s`sp`SCRAM_info`configs(s`tr(r`end_c)`svclvl)
                     (app)), s`tr(r`end_c)`st))) OR
      % the reconfiguration was interrupted
      EXISTS (app: (s`sp`apps)) : s`tr(r`end_c)`reconf_st(app) = interrupted
```

Figure 4. Properties That Define Assured Reconfiguration

CP3 requires that all reconfigurations complete within their required time bound.

CP4 states that the invariant holds if no application generates a signal while preparing to transition, and that if the latter is the case, a mix of the old invariant and the new invariant holds.

CP5 sets out three alternatives for predicates that can be true at the end of a reconfiguration: (1) the reconfiguration was not interrupted and some application reconfigured; (2) the reconfiguration was not interrupted but no application reconfigured; or (3) the reconfiguration was interrupted—in which case, no guarantees are made apart from those stated above.

As the reconfiguration architecture has been written and refined, we have found a significant number of flaws when we have been unable to prove certain properties. The different cases that must be addressed in CP4 are an example of this. Generally, one would expect that the invariant for the current configuration will hold at the end of each execution cycle. This is impossible to guarantee during reconfiguration unless: (1) additional signals generated by any application that must reconfigure cannot be addressed between the time the applications are told of the new configuration and the time the transition condition has been met; (2) any possible signals will not disrupt transition; or (3) an explicit rollback or rollforward mechanism exists for the system: a rollback mechanism must be applied to all reconfiguring applications if one generates a signal, or a rollforward mechanism must be applied to the application generating the new signal to ensure that its state is consistent with the new configuration.

7. UAV Example

7.1 Introduction

In order to demonstrate the concepts that constitute our approach and assess their feasibility, we have specified an example reconfigurable system. This section presents an overview of the example; the full specification, TCCs, and proofs can be found elsewhere [29]. The example is a hypothetical avionics system that is representative, in part, of what might be found on a modern unmanned aerial vehicle (UAV). The example system includes four applications:

- A collection of sensors generates simulated values of altitude and heading that would normally be read from the aircraft's environment.

- The flight control system (`FCS`) receives directions on changes in altitude and heading from either the pilot or the autopilot, and computes appropriate commands to send to the control surface actuators to effect the changes.
- The `autopilot` can be programmed with a target altitude or a target heading to maintain, and it will send commands to the FCS based on the aircraft's deviation from the target.
- The `pilot_interface` simulates pilot commands and transmits them to the autopilot.

For each application, only minimal versions of 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 where appropriate.

The system also models three aspects of the environment that can trigger a reconfiguration:

- Electrical power: the hypothetical aircraft's electrical power generation system contains an alternator and a battery. Failure of the alternator causes the aircraft to switch to its backup battery power source, and at that point some computations are curtailed to preserve battery life.
- Rudder: the aircraft's rudder can become stuck in a hard-over position, requiring the FCS to compensate for its inappropriate position.
- Autopilot: the autopilot can experience a failure of the heading control subsystem or a failure of the entire system. While the failure comes from within one of the applications, it is modeled as a part of the environment because it is something that the system itself is unable to control (otherwise the originating fault would be masked).

7.2 System configurations

The example system is designed to operate in nine different configurations. The sensors and pilot interface each have only one configuration and are assumed to be dependable enough that they are not the limiting factor in system dependability. Other applications can be in different configurations, according to the possible failures described above. The system configurations are shown in Table 2 (the sensors and pilot interface configurations are not listed because they are the same in all system configurations, as described below).

Table 2: UAV System Configurations

Configuration	Power	Rudder	Autopilot	FCS
Full Service	alternator	working	fully functional	normal function
Altitude Hold Only	alternator	working	altitude hold only	normal function
Flight Control Only	alternator battery	working working	nonfunctional disabled	normal function normal function
Rudder Hard-Over Left/Right	alternator	hard-over left/right	fully functional	compensating for rudder
Rudder Hard-Over Left/ Right, Altitude Hold Only	alternator	hard-over left/right	altitude hold only	compensating for rudder
Rudder Hard-Over Left/ Right, Flight Control Only	alternator battery	hard-over left/right hard-over left/right	nonfunctional disabled	compensating for rudder compensating for rudder

7.3 Environmental states and transitions

There are three relevant environmental factors in the system:

- Electrical power generation system (`electrics`): Power can be either generated by the alternator or supplied by the battery.
- Rudder status (`rudder`): The rudder can be working properly, stuck hard-over to the left, or stuck hard-over to the right.
- Autopilot status (`autopilot`): The autopilot is able to provide all services, able to provide altitude hold only, or not able to function.

Environmental transitions are limited to only those that degrade aircraft capabilities. No assumptions are made about coincidence of failures.

7.4 System transitions

Permissible system transitions are defined by two predicates: `degraded`, which reflects the environmental transition restriction that repair will not occur during flight; and `appropriate`, which ties specific reconfigurations to specific environmental states.

7.5 Applications

The modular structure of the system applications is as follows:

Sensors and Pilot Interface

The `sensors` application and the `pilot_interface` application have only one configuration each. Because they only represent system functionality, each includes only one simple module.

Table 3: Example Reconfiguration Stages

Frame	Stage	Action	Predicate
1 (start)	Sensors: `interrupted` All other apps: `normal`	Sensors: signal generated All other apps: normal execution	Sensors: invariant All other apps: invariant
2	Sensors: `halting` All other apps: `exec_halting`	Apps anticipate possible reconfiguration	App postconditions
3	SCRAM: `prepare`$(C_t) \rightarrow$ all apps	FCS: prepare to adjust for rudder All other apps: normal execution	FCS: transition condition All other apps: invariant
4 (end)	All applications: `normal`	All applications: normal execution	All applications: invariant

Autopilot

The autopilot has only one module, but this module has three service levels. The first executes the full autopilot function, including both heading and altitude hold capabilities. The second executes altitude hold function only. The third disables the autopilot.

Flight Control System (FCS)

The `FCS` application has three modules. The first performs the basic calculation of values that will be passed to the actuators, based on inputs from either the pilot or the autopilot. The second modifies the output to compensate for a rudder hard-over condition, if one exists. The third transmits the (modified or unmodified) output to the actuators.

The reconfiguration interfaces for the four applications described above, the nine acceptable configurations, and the configuration transitions are specified in PVS. The instantiation has been type-checked against the specification described in Section 5, and the TCCs have been proven.

7.6 An example system fault-tolerant action

In the example instantiation, each AFTA and each SFTA execute as described in Sections 4.3 and 5.3. Consider, for example, an SFTA that is executing in the Full Service configuration when the rudder becomes stuck in a hard-over left position. The sequence of reconfiguration steps for this case is shown in Table 3. First, the system sensors note the failure and generate a signal for the system to reconfigure during Frame 1 of the SFTA. The signal is passed (logically) from the SCRAM to all applications at the beginning of Frame 2. During this frame, the `sensors` application executes its `halt` function, which does not change the application state in case the signal

occurred due to an application software fault. The other applications execute their `exec_halt` functions during this frame, computing normal results but preparing for a possible reconfiguration. Concurrently, the SCRAM computes the configuration to which the system will transition.

At the beginning of Frame 3, the SCRAM notifies the applications that they will transition to meet their application configurations that correspond to the new system configuration, *Rudder Hard-over Left*. Because the FCS will compensate for the hard-over condition, it is the only application that must reconfigure. Thus, the other applications will execute four AFTAs during the single SFTA described here, where three of the four AFTAs are standard AFTAs under the old/new configuration, and the fourth is slightly modified because the application will have computed `exec_halt` or `halt` instead of `execute` in Frame 2. The autopilot's outputs will be ignored by the FCS until the FCS has finished reconfiguring.

During Frame 3, the FCS sets the service level parameters of its modules to calculate output values that are adjusted for the hard-over condition. Our example does not model control gains explicitly and so the FCS meets its precondition at the end of Frame 3; otherwise, it would initialize data during Frame 4.

7.7 Compliance properties

As explained above, the architecture specification is set up so that compliance is proven if the example is type correct. PVS generated 71 TCCs for the example system. Most TCCs were proven with a single command, since the conditions in the example are relatively simple.

The TCC requiring the most complex proof is shown in Figure 5. Its only nontrivial conjunct is the `covering_txns` requirement. This function returns `true` if a specification defines transitions for any possible combination of environmental state and system configuration in which a reconfiguration signal might be generated. The proof required instantiation of a number of subgoals, which was simple but time-consuming because PVS does not handle instantiation particularly well. This difficulty could be overcome easily with a simple enumeration tool to choose appropriate instantiations. In general, assurance of a reconfigurable system requires a detailed

```
% Subtype TCC generated (at line 297, column 16) for proto_SCRAM_table
    % expected type
       %  SCRAM_table(
       %      proto_apps,
       %      extend[speclvl, {sp: speclvl | NOT sp = indeterminate}, bool, FALSE]
       %         (restrict[speclvl, {sp: speclvl | NOT sp = indeterminate}, boolean]
       %                  (proto_speclvl)),
       %      proto_valid_env, proto_reachable_env)
  % proved - incomplete
prototype_reconf_spec_TCC2: OBLIGATION
   FORALL (x: speclvl):
      proto_speclvl(x) IFF
         extend[speclvl, {sp: speclvl | NOT sp = indeterminate}, bool, FALSE]
            (restrict[speclvl, {sp: speclvl | NOT sp = indeterminate}, boolean]
                 (proto_speclvl))(x)
   AND FORALL (x: speclvl):
      proto_speclvl(x) IFF
         extend[speclvl, {sp: speclvl | NOT sp = indeterminate}, bool, FALSE]
            (restrict[speclvl, {sp: speclvl | NOT sp = indeterminate}, boolean]
                 (proto_speclvl))(x)
   AND covering_txns(proto_apps,
         extend[speclvl, {sp: speclvl | NOT sp = indeterminate}, bool, FALSE]
            (restrict[speclvl, {sp: speclvl | NOT sp = indeterminate}, boolean]
                           (proto_speclvl)),
              proto_valid_env, proto_reachable_env, proto_SCRAM_table`txns,
              proto_SCRAM_table`primary, proto_SCRAM_table`start_env)
   AND FORALL (x: speclvl):
        proto_speclvl(x) IFF
         extend[speclvl, {sp: speclvl | NOT sp = indeterminate}, bool, FALSE]
             (restrict[speclvl, {sp: speclvl | NOT sp = indeterminate}, boolean]
                 (proto_speclvl))(x)
   AND extend[speclvl, {sp: speclvl | NOT sp = indeterminate}, bool, FALSE]
          (restrict[speclvl, {sp: speclvl | NOT sp = indeterminate}, boolean]
               (proto_speclvl))(proto_SCRAM_table`primary)
   AND FORALL (x: speclvl):
        proto_speclvl(x) IFF
         extend[speclvl, {sp: speclvl | NOT sp = indeterminate}, bool, FALSE]
             (restrict[speclvl, {sp: speclvl | NOT sp = indeterminate}, boolean]
                 (proto_speclvl))(x);
```

Figure 5. TCC from the Example System

analysis of system execution under all possible failure scenarios, and so formalizing this requirement has not added an analysis burden to the developer; it has only made that burden explicit.

7.8 Implementation

An implementation of an earlier version of this example [28] exists, although the implementation has not been verified since the emphasis of our work is on specification properties rather than verification. The platform upon which the example instantiation operates is 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 instantiation 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.

The implementation was created only as a feasibility check of the reconfiguration strategy. No experiments were run on the implementation, because experimental evaluation of ultradependable properties is infeasible [7].

8. Conclusion

The complexity of many current safety-critical applications, the scope of the environments in which they must operate, and the strictures placed on them by their dependability requirements are increasing the prominence of reconfigurable system designs. This increase in prominence is due to the opportunity those designs present to meet system functionality and dependability goals. With reconfiguration at the core of a system's architecture, only a small number of functions must be ultradependable, and the rest can be ultradependably fail-stop. In the latter case, only error-detection mechanisms have to be assured, reducing the complexity and cost of software analysis in many systems. At the hardware level, fail-stop computers and transactional semantics can be used to ensure reliability of critical functionality, but some processors can be allowed to fail. Allowing some failures can significantly reduce the power, weight, and space requirements of the system—which, for many embedded systems, results in significant cost savings.

Reconfiguration introduces questions of correct operation and assurance of that correct operation, however. For many systems, meeting functional and timing constraints during reconfiguration as well as during standard operation is critical. This work has demonstrated a means through which general properties of reconfiguration can be assured via proof.

Existing approaches to achieving dependability through reconfiguration involve building the main system and then adding the capability to transition to a separate backup. We advocate building a system with the intent of making it reconfigurable, so that reconfiguration is supported by the high-level system structure. The combination of the approach and the supporting infrastruc-

ture for applying it has the potential to change the way designers think about critical software. By distinguishing between desirable functionality and necessary functionality, they can build systems with significantly higher assurance of dependability, while retaining complex functionality that can increase comfort and efficiency.

Architecting a system to be reconfigurable also presents developers with a method to target their analysis efforts. If formal verification against a specification is to be used, for instance, it is clearly most effective to analyze critical functions completely but analyze only error detection mechanisms for noncritical functions. Safe programming and protection shells are examples of techniques for such analysis. The modular structure of the reconfiguration architecture provides an explicit mechanism for application and composition of these more basic analysis methods.

Finally, our approach provides a method to show clearly the overall picture of a dependability argument for a reconfigurable embedded system. Systems can be built using fault tolerance mechanisms, but these mechanisms show only that certain faults can be tolerated by certain pieces of the system. Writing a specification that describes the system's response to specific classes of errors allows a designer to determine whether the overall dependability requirements of that system have been met. Documenting these requirements at a high level of abstraction in the specification also allows experts to determine more easily whether the properties guaranteed by the software are the properties needed to show system dependability.

Acknowledgments

Tony Aiello, Xiang Yin, and Dean Bushey have been very helpful with the example in this work. Dave Evans, Phil Koopman, Jim McDaniel, and Paul Reynolds made many insightful suggestions. This work was sponsored in part by NASA Langley Research Center grants NAG-1-2290 and NAG-1-02103.

References

[1] Anderson, T., and J. C. Knight. "A Framework for Software Fault Tolerance in Real-Time Systems." *IEEE Transactions on Software Engineering* 9(3):355-364, May 1983.

[2] Anderson, T., and R. W. Witty. “Safe programming.” *BIT* 18:1-8, 1978.

[3] ARINC Inc. “Avionics Application Software Standard Interface.” ARINC Spec. 653, Baltimore, MD, 1997.

[4] Bateman, A., D. Ward, and J. Monaco. “Stability Analysis for Reconfigurable Systems with Actuator Saturation.” Proc. American Control Conf., Anchorage, AK, May 8-10, 2002.

[5] Budhiraja, N., K. Marzullo, F. B. Schneider, and S. Toueg. “Optimal Primary-Backup Protocols.” Workshop on Distributed Algorithms, Haifa, Israel, November 1992.

[6] Burns, A., and A. J. Wellings. “Safety Kernels: Specification and Implementation.” *High Integrity Systems* 1(3):287-300, 1995.

[7] Butler, R. W., and G. B. Finelli. “The Infeasibility of Experimental Quantification of Life-Critical Software Reliability.” ACM SIGSOFT '91 Conference on Software for Critical Systems, New Orleans, LA, December 1991.

[8] Cailliau, D., and R. Bellenger. “The Corot Instruments Software: Towards Intrinsically Reconfigurable Real-time Embedded Processing Software in Space-borne Instruments.” Proc. 4th IEEE International Symposium on High-Assurance Systems Engineering, Nov. 1999.

[9] Garlan, D., S. Cheng, and B. Schmerl. “Increasing System Dependability through Architecture-based Self-repair.” *Architecting Dependable Systems*, de Lemos, Gacek, Romanovsky (Eds.), Springer-Verlag, 2003.

[10] Knight, J. C., E. A. Strunk, 2004, “Achieving Critical System Survivability through Software Architectures,” in *Architecting Dependable Systems II*, de Lemos, Gacek, and Romanovsky, eds., Springer-Verlag.

[11] Knight, J. C., E. A. Strunk and K. J. Sullivan. “Towards a Rigorous Definition of Information System Survivability.” DISCEX 2003, Washington, DC, April 2003.

[12] Kopetz, H. “Time-Triggered Real-Time Computing.” IFAC World Congress, Barcelona, Spain, July 2002.

[13] Kopetz, H, and G. Grunsteidl. “TTP-A For Fault-Tolerant, Real-Time Systems.” IEEE Computer, 27(1), 1994.

[14] Leveson, N., T. Shimeall, J. Stolzy and J. Thomas. “Design for Safe Software.” AIAA Space Sciences Meeting, Reno, Nevada, 1983.

[15] Muller, G., M. Banâtre, N. Peyrouse and B. Rochat. “Lessons from FTM: An Experiment in the Design and Implementation of a Low Cost Fault Tolerant System.” IEEE Trans. on Reliability 45(2): 332-340, June 1996.

[16] Mura, I., A. Bondavalli, X. Zang, and K.S. Trivedi. “Dependability Modeling and Evaluation of Phased Mission Systems: A DSPN Approach.” Dependable Computing for Critical Applications, San Jose, California, Jan. 1999.

[17] Neema S., T. Bapty, and J. Scott. “Adaptive Computing and Run-time Reconfiguration.” Proc. Military Applications of Programmable Logic Devices, Laurel, MD, September 1999.

[18] Perrow, C. *Normal Accidents: Living with High-Risk Technologies*. Princeton University Press, 1999.

[19] Porcarelli, S., M. Castaldi, F. Di Giandomenico, A. Bondavalli, 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.

[20] Powell, D. “Distributed Fault-Tolerance: Lessons from Delta-4.” IEEE Micro 14(1): 36-47, February 1994.

[21] Reason, J. *Human Error.* Cambridge University Press, Cambridge, UK, 1990.

[22] Rushby, J. “Kernels for Safety?” *Safe and Secure Computing Systems*, T. Anderson Ed., Blackwell Scientific Publications, 1989.

[23] Schlichting, R. D., and F. B. Schneider. “Fail-stop processors: An approach to designing fault-tolerant computing systems.” *ACM Transactions on Computing Systems* 1(3):222-238.

[24] Sha, L. “Using Simplicity to Control Complexity.” *IEEE Software* 18(4):20-28, 2001.

[25] Sha, L., R. Rajkumar and M. Gagliardi. “A Software Architecture for Dependable and Evolvable Industrial Computing Systems.” Tech. Rept. CMU/SEI-95-TR-005, Software Engineering Institute, Carnegie Mellon University, 1995.

[26] Shelton, C. and P. Koopman. “Improving System Dependability with Functional Alternatives.” Proc. International Conference on Dependable Systems and Networks (DSN 2004), Florence, Italy, June 2004.

[27] Storey, N. *Safety-Critical Computer Systems*. Prentice Hall: Harlow, U.K., 1996.

[28] Strunk, E. A., J. C. Knight, and M. A. Aiello. “Assured Reconfiguration of Fail-Stop Systems.” Proc. International Conference on Dependable Systems and Networks (DSN 2005), Yokohama, Japan, June 2005 (to appear).

[29] Strunk, E. A., and X. Yin. “Assured Reconfiguration: Specification, Proofs, and Example.” Technical Report CS-2005-05, University of Virginia Department of Computer Science, April 2005.

[30] Wika, K.J., and J.C. Knight. “On The Enforcement of Software Safety Policies.” *Proceedings of the Tenth Annual Conference on Computer Assurance* (COMPASS), Gaithersburg, MD, 1995.

[31] Yeh, Y. C. “Triple-Triple Redundant 777 Primary Flight Computer.” Proc. 1996 IEEE Aerospace Applications Conference, vol. 1, New York, N.Y., February 1996.

Author Biographies

Elisabeth is currently a research scientist at the University of Virginia. She received her Ph.D. from the University of Virginia in May 2005, her M.S. from U.Va. in 2002, and her B.S. from Vanderbilt University in 2000. Her research interests are in dependable computing, with a focus on safety-critical software systems. Her major projects are architecting embedded software systems to be survivable, verifying that an implementation has properties set out in a system's formal specification, clarifying the role of natural language in software development, and exploring the interplay between software development artifacts and system assurance cases.

John Knight is a professor of computer science at the University of Virginia. He holds a B.Sc. (Hons) in Mathematics from the Imperial College of Science and Technology (London) and a Ph.D. in Computer Science from the University of Newcastle upon Tyne. Prior to joining the University of Virginia in 1981, he was with NASA's Langley Research Center.

Dr. Knight's research interests are in software dependability. He is currently working on projects in safety-critical embedded systems and the survivability of critical networked applications. Specific research topics include the use of natural language in specification, tool support for comprehensive specification development and analysis, formal verification, assurance arguments, and network survivability architectures.

From 2001 to 2005 Dr. Knight served as Editor in Chief of the IEEE Transactions on Software Engineering, he is a member of the editorial board of the Empirical Software Engineering Journal, and he is the General Chair of the 2007 International Conference on Software Engineering.