

Reconfiguration Assurance in Embedded System Software

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Elisabeth A. Strunk

May 2005

APPROVAL SHEET

This thesis is submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy (Computer Science)

This dissertation has been read and approved by the Examining Committee:

Dissertation Advisor

Committee Chair

Accepted for the School of Engineering and Applied Science:

Dean, School of Engineering and Applied Science

May 2005

Abstract

As software systems continually become larger and more complex, assurance of their critical properties becomes correspondingly more difficult. 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 in many cases implies that system dependability is probably due at least in part to the care taken by experienced developers in the software's design. Careful 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 many software systems, critical properties are only a small subset of all desirable system properties. In this dissertation, I hypothesize that, in most very complex systems, there is often some much simpler subset of functionality over which critical properties can be expressed. Assuring properties over the simpler subset can provide assurance of critical properties over the entire system. In this case, system dependability can be reduced to a guarantee that either the system will function correctly, or the non-critical function will do nothing to interfere with critical system properties.

My work provides a method for constructing systems to be dependably reconfigurable. 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.

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. 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 software function fail-stop, so that the software either works correctly or fails in a way that does not disrupt other applications. Showing that a specific system complies with the architecture's properties implies assurance of reconfiguration for that system.

To illustrate the ideas in this work, my colleagues and I 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.

Acknowledgments

My thanks go first to my family, because without their support I would never have been brave enough to begin this degree, and they have been a constant source of encouragement along the way. I also appreciate Peggy Reed, who can calm me down when I am stressed by work and deadlines, just by being herself. Yuanfang Cai has been a wonderful friend, always delighted and ready with a hug when I find out a paper has been accepted. John Nguyen has been very supportive and very understanding, and has helped me learn to look for points of view other than mine.

I have been blessed with wonderful colleagues in my department. I cannot imagine a better advisor than John Knight. He has always been there when I needed him, and he has taught me how much fun research can be. Billy Greenwell has been a great friend to work with. My research group is a close-knit set of friends, and I think that is in large part due to my professional and social relationship with Billy. He has also been incredibly valuable as a skeptical audience. Tony Aiello, Xiang Yin, and Dean Bushey have been very helpful with the example in this work, and I am excited about working more with them in the future. Kim Gregg has worked long and hard to take care of the details of my professional activities. I could not have asked for more competent, or more dedicated, support than that which I have received from Kim.

My thanks go to NASA in general (including the supporters of this work who funded NASA Langley Research Center grants NAG-1-2290 and NAG-1-02103), and to Kelly Hayhurst and Michael Holloway, specifically. One of the major challenges of research is both supporting and questioning one's work. For me, doing both at once is extremely difficult. While John Knight has backed my ideas and given me confidence to keep working on them, Kelly and Michael have shared their more impartial perspective with me, which has significantly increased my confidence that my research can play an important role in the engineering of critical software systems.

Table of Contents

Chapter 1. Introduction.....	1
Chapter 2. Related Work	6
2.1. Dependability	6
2.2. Halting a system.....	9
2.2.1. <i>Fail-stop processors</i>	9
2.2.2. <i>Safe programming</i>	9
2.2.3. <i>Safety kernels</i>	10
2.2.4. <i>Protection shells</i>	12
2.3. Reconfiguration	13
2.3.1. <i>Survivability in critical information systems</i>	13
2.3.2. <i>Alternative functionalities</i>	14
2.3.3. <i>Graceful degradation</i>	15
2.3.4. <i>Quality of service</i>	15
2.3.5. <i>Performability</i>	16
2.3.6. <i>Simplex architecture</i>	16
2.3.7. <i>Fault tolerance</i>	17
2.3.8. <i>Reconfiguration in practice</i>	17
2.3.9. <i>Reconfiguration for reasons other than failure</i>	18
2.4. Software architectures to aid assurance arguments	18
2.5. Summary	19
Chapter 3. Reconfiguration as an Architectural Driver	21
3.1. Current strategies for arguing dependability	21
3.1.1. <i>Fault avoidance</i>	22
3.1.2. <i>Fault elimination</i>	23
3.1.3. <i>Fault tolerance</i>	25
3.2. Problems arising from complexity.....	27
3.3. Bounding the dependability problem.....	28
3.4. Reconfiguration for embedded systems.....	29
3.5. Reducing resource requirements.....	31
3.6. Dealing with COTS components and legacy software	32
Chapter 4. Informal Model	34
4.1. Reconfigurable fail-stop systems.....	34
4.2. Informal definition and system assumptions	35
4.3. Reconfigurable fault-tolerant actions.....	37
4.3.1. <i>Fault-tolerant actions</i>	37
4.3.2. <i>AFTAs</i>	38
4.3.3. <i>SFTAs</i>	39
4.4. Real-time guarantees on fault-tolerant actions	40
4.5. Candidate reconfiguration architecture.....	42
4.5.1. <i>Application reconfiguration</i>	42
4.5.2. <i>System reconfiguration</i>	45
4.5.3. <i>Example implementation platform</i>	46

4.6. Summary	48
Chapter 5. Formal Model.....	50
5.1. The Prototype Verification System and the proof structure	51
5.1.1. <i>The PVS language</i>	52
5.1.2. <i>The PVS system</i>	52
5.1.3. <i>Reconfiguration proof structure</i>	53
5.2. Formalism assumptions	54
5.3. Application structure.....	55
5.3.1. <i>State</i>	56
5.3.2. <i>Modules</i>	58
5.3.3. <i>Applications</i>	59
5.4. System model and the SCRAM.....	60
5.4.1. <i>Operating environment</i>	60
5.4.2. <i>System configurations</i>	62
5.4.3. <i>System transitions and the SCRAM</i>	63
5.5. State traces	64
5.5.1. <i>System state</i>	66
5.5.2. <i>Application execution</i>	66
5.5.3. <i>Environmental transitions</i>	68
5.5.4. <i>State trace</i>	69
5.6. Reconfiguration definition	69
5.7. Architecture instantiations	73
5.8. Summary	76
Chapter 6. UAV Example.....	77
6.1. Introduction.....	77
6.2. System configurations.....	79
6.3. Environmental states and transitions	79
6.4. System transitions	81
6.5. Applications	81
6.6. An example system fault-tolerant action	82
6.7. Compliance properties	84
6.8. Implementation	86
6.9. Summary	87
Chapter 7. Conclusion, Contributions, and Future Work	88
7.1. Conclusion	88
7.2. Contributions	90
7.3. Future work.....	92
Bibliography	97
Appendix A: Architecture Specification.....	A - 1
State and Operations	A - 2
Module Framework.....	A - 3
Application.....	A - 4
SCRAM	A - 6
Reconfiguration Specification	A - 8
Monitor	A - 10
Trace	A - 13

Lemmas	A - 15
Invariant Lemmas	A - 18
Reconfiguration Properties	A - 20
Appendix B: UAV System Specification.....	B - 1
Example State	B - 2
Environment.....	B - 4
Sensors	B - 6
Pilot Interface.....	B - 9
Flight Control System Functional Specification.....	B - 11
Flight Control System.....	B - 14
Autopilot Functionality.....	B - 18
Autopilot	B - 21
Prototype System Reconfiguration Specification	B - 24
Appendix C: UAV System Proof Obligations	C - 1
Appendix D: Example Property Proof	D - 1

CHAPTER 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 in many cases implies that system dependability is probably due at least in part to the care taken by experienced developers in the software's design. Careful development and review are likely to become less effective as the complexity of the developed software grows beyond the limits of straightforward human comprehension.

My work provides a method for constructing systems to be reconfigurable. 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 employ various techniques to provide some professed argument of assurance. Controlled development processes and extensive testing are the most common of these techniques. There is no scientific evidence to support the claims of a high level of effectiveness of the former, and the scientific evidence surrounding the latter shows that it is infeasible [14]. Stronger analysis mechanisms, such as model checking and correctness proofs, possess their own limitations. Model checking is effective in showing properties at the algorithmic level, but it uses a model of the analyzed software and so leaves gaps in its assurance of implementation properties. Correctness proofs, while able to provide high levels of assurance, are generally dismissed as being impracticable for any but relatively simple systems.

In hardware dependability, replication has historically been sufficient because hardware faults were primarily degradation faults, and so hardware components failed independently. Independent failures allowed replicated components' failure probabilities to be multiplied and very low overall failure rates achieved. There has been no evidence to date, however, that software errors occur independently [25]. Even in hardware, design faults become more prominent as complexity increases, and so independence cannot always be assumed. Thus, some alternative analysis technique is necessary.

While it is unclear that independence of software failures can be shown, it is reasonable to assume that the community can continue to build systems of current complexity with similarly low failure rates. The problem with system complexity thus

reduces to the ability to show that the additional function added to take advantage of progress in other disciplines 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 [12]. Design practices often take into account differences in criticality and exploit them, particularly in the form of simpler software backups or electromechanical backups. An example is the Simplex architecture proposed by Sha *et al.* [45], which employs this strategy not only to aid dependability arguments but also to enable dependable implementation of online software upgrade and the use of Commercial Off-The-Shelf (COTS) components. Research in facilitating such reconfiguration has been conducted and operational systems exist that are capable of reconfiguration, such as the Boeing 777 flight control system [56].

Previous research has taken an existing system and provided the capability to reconfigure to some alternative service as an addition to the system's original architecture. My work starts with the assumption that building a non-reconfigurable system is impossible for complex systems that must meet very high levels of assurance. 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 function that is allowed to fail, and easier to assure, since my reconfigurable architecture is explicitly designed to support assurance arguments.

This work also 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. Ad-hoc reconfiguration assurance is insufficient in systems whose dependability arguments are centered on their reconfiguration properties. The actions involved in reconfiguration must be completed correctly, on time, and with very high confidence in correct operation.

This work sets forth an approach to system construction to ensure dependability properties by ensuring 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 software function fail-stop [43], so that the software either works correctly or fails in a way that does not disrupt other applications. Showing that a specific system complies with the architecture's properties implies assurance of reconfiguration for that system.

To illustrate the ideas that I describe, my colleagues and I 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 an application to read and report simulated sensor data, an application to simulate pilot commands, a flight control application, and an autopilot. The parts of these applications that are relevant to my research have been implemented, although the functionality is merely representative.

This dissertation is organized as follows. Chapter 2 discusses related work on dependability, reconfiguration, and architecture formalisms. Chapter 3 elaborates the reasons reconfiguration is needed to build dependable systems, and the advantages such a

strategy provides. Chapter 4 presents an informal discussion of the architecture itself, and Chapter 5 gives an overview of the formal structure (the entire formal architecture specification can be found in Appendix A). Chapter 6 describes an example avionics system that instantiates the architecture (the full specification for the example can be found in Appendix B). Finally, Chapter 7 lists the contributions of the work, its implications for software engineering, and future research directions to aid in realistic application of this technology.

CHAPTER 2

Related Work

The method of constructing a system to meet its dependability requirements presented in this work involves: (1) assuring that the system's primary function will halt safely and in bounded time if a reconfiguration is signaled, and (2) assuring that the system will reconfigure properly once it has halted. It does this through analysis mechanisms supported by its architecture. The accepted meaning of dependability, related work in both halting and reconfiguring a system, and related work in assurance at the architecture level are discussed in this chapter.

2.1. Dependability

Many terms have been applied to safety-critical software systems in an informal sense. One of the most common is *reliable*, the everyday meaning being that one relies on a system to do what is intended of it. Hardware reliability is generally defined in a stochastic manner: it is allowed to fail, but only with a very low probability. Such a probabilistic metric can be applied to software, but this usually is not the most appropriate metric to

use, since the stochastic properties of digital systems can be significantly harder to determine than those of analog systems.

Avizienis, Laprie, and Randell have created a taxonomy of facets of what they term *dependability* [7]. Dependability in their sense, “the ability to deliver service that can justifiably be trusted,” replaces the informal notion of reliability but expands on this notion to include other facets that must be ensured for many systems if the systems’ users are to depend on them. This taxonomy has become a *de facto* standard as well as a *de jure* standard in progress through International Federation for Information Processing (IFIP) Working Group 10.4 [17]. The facets Avizienis *et al.* define are [7]:

- **availability**: readiness for correct service,
- **reliability**: continuity of correct service,
- **safety**: absence of catastrophic consequences on the user(s) and the environment,
- **confidentiality**: absence of unauthorized disclosure of information,
- **integrity**: absence of improper system state alterations,
- **maintainability**: ability to undergo repairs and modifications.

These properties are in some sense orthogonal to the function specified for the system. For instance, availability is the probability that the system will be able to provide service at time t , i.e., its requirements are met with a certain probability at the time when it is called.

Butler and Finelli coined the term *ultrareliability* to correspond to applications with a failure rate that must be less than 10^{-7} failures per hour [14]. The term *ultradependability* has come to represent similar failure probabilities of dependability as defined in the taxonomy of Avizienis *et al.*

An example embedded system that must be ultradependable is the flight control system on a commercial aircraft. The US Federal Aviation Administration (FAA) has a carefully-written set of precise requirements for development of critical aircraft software [40]. Certification requirements define what regulatory agencies see as the established state of the practice in development of critical software, and so I include a short description here.

The FAA categorizes aircraft function into three criticality levels—minor, major, and catastrophic—according to the potential severity of its failure conditions [18]. Failure conditions must have probabilities occurrence that are inversely proportional to their potential consequences: “(1) Minor failure conditions may be probable. (2) Major failure conditions must be improbable. (3) Catastrophic failure conditions must be extremely improbable.” “Probable” is defined as “anticipated to occur one or more times during the entire operational life of each airplane”; “improbable” as “not anticipated to occur during the entire operational life of a single random airplane”; and “extremely improbable” as “so unlikely that [the failure condition is] not anticipated to occur during the entire operational life of all airplanes of one type.” Quantifying these definitions leads to probabilities that can be extremely small. “Extremely improbable”, for example, corresponds to a failure rate of 10^{-9} per hour of operation.

Generally, the critical systems I address in this work must be ultradependable. The assurance mechanisms can contribute to construction of less dependable systems as well, however, since software with few defects is more desirable than software with many defects regardless of the consequences of failure.

2.2. Halting a system

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.

2.2.1. Fail-stop processors

Schlichting and Schneider documented the concept of a fail-stop processor 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 fail-stop semantics [43]. They define a fail-stop processor to consist of one or more processing units, volatile storage, and stable storage. The failure semantics of a fail-stop processor are [43]:

FS1: It stops executing.

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

Stable storage may not be affected by a failure.

An embedded system of the type assumed for my work 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.

2.2.2. Safe programming

The software analog of fail-stop machines is the concept of *safe programming* introduced by Anderson and Witty [4]. Safe programming requires (in part) modification

of the postcondition for a program by adding an additional clause allowing the program to terminate without modifying 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 rather than assuring overall functionality.

2.2.3. Safety kernels

Related to safe programming is the idea of a *safety kernel*. The idea has been studied by a number of researchers. Leveson *et al.* use this term to describe a system structure where mechanisms aimed to achieve safety are gathered together into a centralized location [30]. A set of fault detection and recovery policies specified for the system is then enforced by the kernel. Error detection is carried out using safety assertions and a module timer in the kernel, and the recovery policy is enacted by either changing module priorities or initiating new modules. The kernel also provides support for human operators to choose an appropriate recovery strategy.

Rushby has more clearly defined the role of a safety kernel as a small component that guarantees the enforcement of properties where two conditions hold. First, each system-level property of interest must be able to be expressed at the kernel level. Second, each property must be expressible using a second-order assertion which, informally, states that every operation over which P is quantified is ultimately effected through calls to kernel functions, and the kernel itself cannot be modified [42]. Second-order assertions define conditions that should always hold and are particularly well suited to describing actions that should never occur (negative properties). Enforcement of positive behaviors is much

more doubtful because it is difficult to ensure the proper use of functions that are provided.

Wika and Knight give a characterization of classes of safety policies that might be enforced [55]. They also introduce the idea of *weakened properties*, properties that are not checked in the kernel, but which the kernel ensures are checked by the application. This might be appropriate when policies are difficult to enforce without including a good deal of function in the kernel. Using weakened properties does not give the same level of assurance as using kernel-enforced properties, but it gives higher assurance than provided by the application alone while saving on the cost and assurance complexity that would result from requiring significant computational effort in the kernel.

Burns and Wellings build on Leveson's and Rushby's work to define a safety kernel as a *safe nucleus* and a collection of *safety services* [13]. The safe nucleus manages safety properties of the computing resources on which a set of applications will run, e.g., memory partitioning and network management. The safety services check safety and timing invariants of individual applications. The safety services evade the problem described by Rushby of enforceability of only negative properties; including them means that all computation requests of an application can be monitored to check that safety assertions hold. A liveness property, for instance, could be ensured by the safety service's calling a particular application function if it has not seen an execution of that function within bounded time. Note that in the framework outlined above, the safe nucleus would be incorporated into the network and operating system layers (which are not discussed further here), and it is the safety services that are provided by the reconfiguration framework.

Peters and Parnas [36] use the idea of a *monitor* that checks the physical behavior of a system by comparing it with a specification of valid behaviors. They explore the imprecision in a system's ability to detect its precise physical state and how this relates to the properties that must be checked by the monitor. They distinguish between an optimistic monitor, which signals an error only if there is no valid system behavior that could lead to a perceived system behavior, and a pessimistic monitor, which signals an error if there is any invalid system behavior that could lead to an observed behavior. These ideas apply to the safety kernel concept as well, as there may be some slack in safety policies or known imprecision in the physical system.

2.2.4. Protection shells

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 certain properties about their associated software components by checking that certain properties of those components' outputs hold, in a manner similar to the monitors described by Peters and Parnas [36]. Less critical components, then, can have shells that guarantee the entire piece of function is fail-stop, while crucial components' shells can ensure fail-operational capability. Whichever capability is needed, the analysis associated with the shell will be much simpler than that associated with the full system because the shell for each component is much simpler than the component itself and is explicitly designed to facilitate that analysis.

2.3. Reconfiguration

2.3.1. Survivability in critical information systems

In large networked systems, reconfiguration in response to failures has come to be 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 [27]; the system will “survive” (i.e., continue some operation), even in the event of damage. For many information systems, particularly critical infrastructure systems such as the financial payment system and electric power grid, it is claimed that ensuring strict dependability properties is impractical. Not only is it difficult to keep strong centralized control over them, but the necessary redundancy to ensure their dependability through masking would be too expensive to implement [27].

Knight *et al.* give a definition of information system survivability based on specification: “A system is survivable if it complies with its survivability specification” [27]. They draw on the properties mentioned above and present a specification structure that tells developers what survivability means in an exact and testable way. When followed, this structure will cause them to document what it means for their system to be survivable.

For networked systems, the loss of a single component or even a moderate number of randomly-distributed components creates only a minor disturbance and must be expected. Reconfiguration at the network level would be employed only in the event of damage with significant consequences such as very large numbers of failures, or if moderate numbers of failures suggested a common cause. The main challenge in effecting reconfiguration in these systems is managing system scale.

Embedded software has certain similarities to and differences from large networked systems. It has a certain level of intellectual manageability stemming from its less distributed nature. However, it still rarely possesses the qualities of a stand-alone application. Embedded systems generally receive input from and send output to other hardware devices or software applications. These devices can fail just as network nodes can, and such failures must be considered in the construction of dependable systems.

Furthermore, safety-critical embedded systems are likely to have hard real-time requirements, their dependability requirements are likely to be much tighter than those for networked information systems, and the allowance for duplication much smaller. If one ATM fails, a banking customer can use another; if it takes longer than expected on occasion, this is merely irritating. Only large numbers of such failures can cause significant problems. The smaller scale of embedded systems facilitates their analysis, but it makes them more tightly coupled and thus necessitates deeper rigorous analysis.

Finally, many embedded systems require some minimal level of function to ensure safety. For example, software controlling aircraft flight cannot simply terminate in midair; there must be some basic level of operation that it is guaranteed to maintain. Networked systems are likely to see a more gradual degradation, with the boundary between effectiveness and ineffectiveness being blurred.

2.3.2. Alternative functionalities

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 [46]. They have created a model of system utility based on user requirements for system service, and a method to determine which sets of services might be feasible after

different combinations of failures. Their work is focused on reconfiguration requirements rather than effecting the reconfigurations themselves, and so complements my work by assisting domain experts in determining the different forms of service a system might offer when failures occur.

2.3.3. Graceful degradation

One definition of graceful degradation, from the telecommunications industry, is: “Degradation of a system in such a manner that it continues to operate, but provides a reduced level of service rather than failing completely” [54]. According to this definition, graceful degradation could be accomplished by reconfiguring to some other operational specification when a failure occurs. Generally, graceful degradation refers to provision of maximum possible utility given a certain set of working functional components, which means it provides fine-grained control of functionality during degradation. A gracefully degrading system is likely to degrade in steps as resources are lost, but those steps are not necessarily calculated explicitly at design time. My reconfiguration architecture, on the other hand, sacrifices some of graceful degradation’s postulated utility in order to permit rigorous but practical analysis of reconfigurable software. Graceful degradation, then, might be more appropriate to classes of systems where provision of function is not critical, and complete failures from unforeseen interactions are acceptable on occasion.

2.3.4. Quality of service

The telecommunications industry also has a definition for quality of service: “Performance specification of a communications channel or system. Note: QOS may be quantitatively indicated by channel or system performance parameters, such as signal-to-

noise ratio (S/N), bit error ratio (BER), message throughput rate, and call blocking probability” [54]. Quality of service (QoS) is similar to graceful degradation in that it focuses on providing the most value given available resources and, like my reconfiguration architecture, does this by incorporating some number of discrete functional levels. However, the term is generally used to refer to specific aspects of a system, e.g., video quality or response time. QoS could be used by a reconfigurable system, but my architecture permits a much broader impact on system function, changing the function more dramatically or replacing it altogether.

2.3.5. Performability

The concept of performability is related in a limited way to reconfiguration [33]. A performability measure quantifies how well a system maintains parameters such as throughput and response time in the presence of faults over a specified period of time [34]. Thus performability is concerned with analytic models of throughput (response time, latency, etc.) that incorporate both normal operation and operation in the presence of faults but do not include the possibility of alternative services.

2.3.6. Simplex architecture

The Simplex architecture of Sha *et al.* [45] uses a simple backup system to compensate for uncertainties in a more complex primary system. The architecture was developed to facilitate use of COTS components in safety-critical systems, rather than being a general software construction strategy. It assumes the existence of two major functional capabilities that have some application-level design difference between them (*analytic redundancy*). The examples given primarily involve control systems; whether

two software systems can be constructed so that they fail independently is still an open question [25]. My reconfiguration architecture is similar, but: (1) takes a more general view of what user requirements might be, allowing more than two distinct functions; (2) uses tighter component control, disallowing component replacement (as opposed to component reconfiguration) online in order to facilitate stronger analysis; and (3) addresses the problems associated with non-independence of software failures by integrating different, simpler functionality with a system's standard primary function.

2.3.7. Fault tolerance

Fault tolerance is a mechanism that can be used to achieve required dependability properties by coping with faults that remain (or arise) in a system once it is deployed. Generally, fault tolerance is used to describe mechanisms through which faults are masked completely. Reconfiguration in response to failures is a form of fault tolerance, but it is designed to allow a system to tolerate faults by transitioning to provision of alternative service, rather than by making the faults through redundancy.

2.3.8. Reconfiguration in practice

The notion of reconfiguration to deal with faults has been used extensively in safety-critical and mission-critical systems. For example, the Boeing 777 uses a strategy similar to that advocated by Sha's Simplex architecture in which the 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 [44].¹ The Airbus A330 and A340 employ a similar strategy [47], as have embedded systems in other critical domains. Existing approaches to

1. Sha claims that it uses control laws of the 747, but I was not able to find corroboration of this claim.

reconfigurable architectures are, however, ad hoc; although the system goals are achieved, the result is inflexible and not easily assured.

2.3.9. Reconfiguration for reasons other than failure

Reconfiguration can also aid the construction of software systems where the system must be dependable but reconfiguration does not support this explicitly. Such systems include nuclear power plants, where the system can be in operational or maintenance phases; and 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 [32]. 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 [15]). Reconfiguration also appears in “intelligent” control systems (such as described by Stewart *et al.* [48] and Bateman *et al.* [8]), and in the more general case of adaptive reconfigurable computing [35].

2.4. Software architectures to aid assurance arguments

For large, complex systems, showing that an implementation has the characteristics outlined above can be a daunting task. The software architecture community has worked towards architecture formalization as a means of assurance for properties of the architecture, most notably component synchronization [2]. Dynamic software architectures, or architectures that permit high-level runtime reconfiguration, are a field in their own right [10]. While combining my architecture with existing architectural analysis techniques is an interesting direction for future research, existing techniques do not apply

directly to my work for two major reasons. First, while the techniques are generally able to express temporal properties of application interaction, little work exists in assuring hard real-time properties. Second, architecture description languages and analysis techniques are targeted at a layer of abstraction above that which I am targeting. General architecture analysis mechanisms are able to show properties of general classes of architectures, while my architecture analysis is for my particular architecture. Likewise, my architecture describes a dynamically reconfigurable system, but it is not a dynamically reconfigurable architecture. The latter would have components whose interactions change at run-time, while my architecture allows only functionality to change at run-time, although of course interaction change could be simulated by creating applications that only provide service in certain configurations.

Garlan *et al.* [20] have proposed the use of software architectural styles as a general method of error detection and reconfiguration execution to improve dependability. Such a strategy could be very helpful in enforcing system-level properties that cannot be described by conjunction of component properties. They do not present a method of assuring their styles, however, and so an exploration of the interaction of their work and mine is left for the future.

2.5. Summary

In summary, researchers have addressed a variety of topics related to the research documented in this dissertation. Topics range from definitions to building blocks to informal reconfiguration approaches. My research takes a completely different approach in which reconfiguration is the heart of the way in which dependability is achieved. None

of the previous research tackles the problem in this way, and none provides the comprehensive, assured approach presented here.

CHAPTER 3

Reconfiguration as an Architectural Driver

The goal of this work is the creation and assurance of an architecture within which a reconfigurable system can be built. The theory is a general one, and so reconfiguration can happen for any reason. The most pressing reason in the field of critical systems, however, is that of fault tolerance, and so the prime motivation for the work is to enable a software system to deal with errors. This chapter elaborates the importance of reconfiguration in critical software systems. It summarizes mechanisms that are currently used to achieve software system dependability, difficulties in assuring systems to ultradependable levels even when using state-of-the-art analysis techniques, and how the use of reconfiguration as the basis of a software architecture attacks the heart of those difficulties.

3.1. Current strategies for arguing dependability

Many techniques have been developed to support the engineering of dependable systems. In a broad sense, these techniques fall into three primary areas: fault avoidance,

fault elimination, and fault tolerance. Combined with system analysis techniques such as fault-tree analysis, these approaches to dealing with faults permit useful dependability predictions to be made about specific designs. However, none of these techniques effectively addresses the dependability problems arising from the growing complexity of embedded software. This section describes several strategies for building dependable software systems, explaining some of their limitations.

3.1.1. Fault avoidance

According to Avizienis, Laprie, and Randell, fault avoidance, listed in their taxonomy as fault prevention, “is attained by quality control techniques employed during the design and manufacturing of hardware and software” [7]. Organizations commonly use their control over development processes to justify claims of safety of a product; see, for example, the Federal Aviation Administration’s guidelines on software [40]. While some correlation between software quality and process control probably exists, there is no evidence to suggest that processes alone can ensure ultradependability. Even the people who define the processes do not claim that those processes will meet dependability requirements. They simply use processes in this way as a “best-effort” strategy.

Formal specification and analysis are development practices specifically designed to support claims on dependability properties of software systems. There is evidence that the majority of safety-critical software errors result from incorrect requirements [31]. Much of the difficulty in requirements capture stems from the application domains in which systems are to operate and thus is outside the scope of computing research. Part of the difficulty, however, stems from flawed communication between domain experts who understand the requirements and developers who implement them. Formal specification

(such as described in Potter, Sinclair and Till [39]) has many advantages, including potential for static analysis (see below), but one of its primary benefits is to ensure software requirements are stated in a precise and unambiguous manner to avoid errors and inconsistencies in implementations. Formal analysis can ensure that certain abstract properties hold over a formal specification, thus increasing confidence in the specification. However, the fundamental problem of formalizing requirements correctly still remains.

3.1.2. Fault elimination

Fault elimination is the process of removing faults that have been introduced into a system. Examples of techniques for fault elimination include traditional testing, static analysis, and dynamic analysis.

Testing

Another argument for placing confidence in controlled development processes hinges on testing. Testing has the advantage of being a formal process (when the results are compared to specific expected outcomes), but also possesses the disadvantage of being unable to provide full confidence in most large systems. Various test metrics, such as 100% statement coverage and modified condition/decision coverage [21], are used, but while these ameliorate the problem, no algorithm for test selection other than exhaustive testing can guarantee that its output will always reveal the presence of errors [23]. Furthermore, in nonterminating concurrent systems, it is impossible to test all possible combinations of operation sequences. Even statistical analysis of test results for such

properties is insufficient to show properties at ultradependable levels [14]. Finally, there can sometimes be a question of whether the system output meets the test criteria [11].

Static Analysis

A plethora of tools and techniques has been created to analyze software before it is run. Familiar forms of static analysis include syntax and type checking; these are often required for any development in particular high-level languages. Tools for lightweight formal methods allow users to invest somewhat more effort in programming in order to derive proportionally more assurance from analysis. Such approaches can establish important, but limited, properties of software systems. Stronger formal verification of the correspondence between an application and a specification, and putative theorem proving about a specification, can provide strong guarantees of a wide range of valuable properties but require significant effort.

Dynamic Analysis

It is also possible to check software properties while the software executes. Certain classes of type constraints are enforced this way, particularly in object-oriented languages where exact types might not be known until run-time. Low-level properties such as array-bound constraints can also be enforced dynamically. Dynamic analysis can be used to infer properties of a program by monitoring program execution sequences, but there are generally limitations on that inference, since dynamic analysis cannot cover all possible execution sequences any more than traditional testing can.

Although combinations of these analysis mechanisms can provide a significant level of confidence of proper system behavior, none of them is strong enough to give

ultradependable assurance over an entire software product. In order to prove formally that software is *correct* in the common sense (rather than in the sense of compliance with a specification, as in partial or total correctness), one must prove that the formal model on which the software is based is a true depiction of the software's operating environment. Unfortunately, the operating environment is informal, and doing formal analysis on any sort of informal artifact is impossible. Informal analysis can give some assurance of such properties, but informal analysis is conducted primarily by humans, who have difficulty in understanding large, complex systems in their entirety without specific support for doing so. Thus, it is useful to put measures into place to look for and deal with faults that might still remain in an operational system.

3.1.3. Fault tolerance

Fault-tolerance mechanisms detect and deal with faults that remain in a system after it is deployed. Examples of fault-tolerance mechanisms include non-software backups, error recovery, and software fault-tolerant architectures.

Non-software backups

Hardware backup systems or other technologically diverse approaches are a way of circumventing the problems in pure software solutions. Most aircraft with fly-by-wire control systems also have electromechanical control systems that can be employed in the event that the fly-by-wire system malfunctions. Some also have hardware restrictions that will not allow unsafe commands from the computer to be executed. Many medical devices, such as pacemakers, have a separate hardware system that can take over to provide basic critical function in the event of software failure. While these hardware

backups work well in preventing system failures due to software errors, they are expensive and cannot always provide as much functionality as is possible with a software solution. These issues have caused mechanical backups to become less common; some newer aircraft models, for example, do not carry fully-functional hardware alternatives (e.g., the Airbus A320 [1]). In military aircraft this is often because maneuverability requirements leave the aircraft aerodynamically unstable; under such circumstances, pilot response time is not good enough to fly the aircraft and thus mechanical backups would be useless [16]. In civil aircraft, the removal of the backup systems is often a matter of economics.

Error recovery

Two common error recovery strategies are rollback and rollforward. Rollback mechanisms periodically save consistent state, and return the system to that state if the system produces an error. Rollforward mechanisms attempt to deal with the error by either computing a new consistent state or transitioning the system to a predetermined consistent state. These forms of error recovery are typically used as fault masking techniques, i.e., they correct system state so that the error does not affect other system components.

Software fault-tolerant architectures

Design diversity attempts to deal with software defects at run time by using independently-developed software replicates. For instance, recovery blocks allow execution of a portion of a program to be checked; if the output does not satisfy correctness criteria, an alternate block can be used instead [22]. N-version programming is similar to recovery blocks in that several different versions of a program are written, but an even more explicit analogue to hardware fault tolerance in that those versions are run in

parallel and their results voted on rather than being evaluated serially by a dedicated output-checking mechanism [6]. Unfortunately, such forms of software fault tolerance do not produce the same benefits in software that they do in hardware. Because software designs are so complex, it is currently indicated that there is no way to show that they will fail independently [25]. Without independence guarantees, failure probabilities of the different versions cannot simply be multiplied to calculate the composite dependability numbers, and so assurance at ultradependable levels is infeasible.

Analytic redundancy [45] takes advantage of design diversity as determined by the software application in implementing fault tolerance. For example, a system using simple control laws can be used as a backup to a system with more complex control laws [44]. The type of diversity it advocates is independent by nature, so that the issues in showing version independence do not apply. While this method promises to be very effective where applicable, the extent to which it can be used in general software systems is not clear.

3.2. Problems arising from complexity

The above approaches to dealing with faults permit useful dependability predictions to be made about specific designs. However, none of these techniques effectively addresses the dependability problems arising from the growing complexity of critical software. Software has many touted advantages such as flexibility and economy, and these make its use tempting for 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 [41]. When functioning properly, automation can greatly increase the reliability and safety of such

systems. It is often able to increase the capabilities of the systems beyond the limits set by the abilities of human operators, such as in the aerodynamically unstable aircraft mentioned earlier. Such functionality 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 becomes 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 easily have unacceptable consequences.

Introducing additional safety checks adds some assurance that the system will function as desired, but it also further complicates the system and thus adds its own risks [37]. Limiting the functionality included on a processor is an infeasible option because not only is it a poor choice economically, in practice it is likely to be ignored. The best we as a community can hope for is to bring the problem of program comprehension to a manageable level. While such a strategy does not solve the problem, it helps the humans who build and analyze safety-critical software to understand the software well enough to determine whether it satisfies its safety criteria.

3.3. Bounding the dependability problem

In many cases, much of the functionality included in a system is not required for system safety. For example, the autopilot system on a commercial aircraft could contribute to an accident, but while its dependable operation is a significant part of the safety case for the aircraft, cessation of its function is unlikely to have catastrophic consequences as long

as the pilot is informed. This suggests that such a system does not need to be *ultradependable* as much as it needs to be *fail-stop* [43]: either the autopilot works correctly, or it stops and alerts the pilot. The complete avionics system for the aircraft needs to be able to operate without the autopilot (and other similar subsystems) so that safety is not compromised if the autopilot fails. In some cases, the remaining functional elements of the system will alter their modes of operation to compensate for the lack of function in the failed subsystem. In this way, reconfiguration at the system level can compensate for application faults. In an informal sense, such an avionics system is *survivable* [26] rather than *ultradependable*.

The emphasis in safety-critical systems has always been to mask the effects of faults, but that is becoming increasingly difficult as the complexity of the systems increases. Exploiting reconfiguration for these systems can reduce complexity of critical function and limit the amount of software that is crucial to dependable operation.

3.4. Reconfiguration for embedded systems

A comprehensive dependable reconfiguration framework for critical embedded systems must address a number of system characteristics:

System Timing and Resource Constraints

Embedded systems are often severely limited in power, cost, space, and weight, and so they are tuned to take the best possible advantage of their underlying resources.

System Coupling

System components frequently have a very strong dependence on one another, and so the state of various system components must be seen individually and also as a whole when determining appropriate system behavior.

Damage and Repair Sequences

Failures can occur in sequence, and while sometimes the system size allows reinitialization, at other times system criticality precludes any service interruption.

Heterogeneous Criticality

Criticality of embedded system services also varies with function and with time. A medical application, for instance, is generally less dangerous during surgical planning than during the surgery itself.

Complex Operational Environments

While the operational environments of embedded systems are relatively localized, they can still be affected by a variety of factors. Avionics systems, for example, are affected by factors such as weather, altitude, and geographic location.

Building reconfiguration into the basic design of a system allows all of these factors to be considered while system requirements are being developed. The distinction between crucial and noncrucial function can be made clearly at the requirements level and set forth in the system's specification. This allows designers to specify the properties they want, rather than employing individual mechanisms that work towards ensuring desired properties (e.g., fault tolerance for certain portions of the software).

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. The approach I have taken in constructing a reconfiguration framework is unique in that it is designed explicitly to facilitate rigorous formal system analysis based on a system's reconfiguration capability.

3.5. Reducing resource requirements

Achieving hardware dependability through pure replication is nontrivial now and will be very difficult to achieve in the future because system component failures will become more common as the number of components increases. Even though lower component cost means that those components can be more easily replicated, replication—and the environmental shielding, power, space, 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 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 planned operation 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 number needed to provide full service. Though not impossible, 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.

In addition to facilitating software dependability arguments, reconfiguration offers a trade-off between functionality and hardware resources. This trade-off is exploited by building elements of the system (such as those implementing the primary functionality) with less provision for coping with faults than normally might be preferred. 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 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. This saves power, weight and space.

3.6. Dealing with COTS components and legacy software

Reconfiguration also can potentially ease the use of standardized or legacy software in critical systems. Use of software that is not specifically developed to be part of a particular system is often difficult, because its interactions with the rest of the system can be unclear. In avionics software, for example, extensive testing at both the unit and system levels must be performed to specific, comprehensive coverage metrics, and this is true of reused software as well as new software. Dead code is disallowed, and deactivated code must be treated with care, if the software to go on an aircraft is to be certified; many COTS products do not meet such requirements.

Some commercial-off-the-shelf (COTS) software is built with certification in mind, so that many of the software tests can be reused across several certified systems. In general, however, COTS software can be built with very weak dependability goals. Even with

dependable COTS components, porting software from one environment to another can still incur major costs for customization (and possibly certification) to deploy it in the new environment. Reconfiguration could allow developers to thoroughly test the simpler code and error-detection code for the new environment, but only check the majority of the software at runtime. Such a scheme would significantly reduce pre-deployment testing requirements.

CHAPTER 4

Informal Model

The mechanism through which embedded system reconfiguration is achieved is complex, and going about its development in an ad hoc way could reduce dependability rather than increase it. Furthermore, critical system function must often be achieved with very high assurance, and high levels of assurance require a precise statement of the properties that must be assured. This chapter presents an intuitive picture of the properties that a reconfigurable system must exhibit, and an informal discussion of an architecture that guarantees those properties. Chapter 5 formalizes these concepts.

4.1. Reconfigurable fail-stop systems

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 protocol 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 to mask the 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 the action or by some alternative means. My 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 possibly *different* function. An FTA in this 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 function (although possibly not the one requested).

4.2. Informal definition and system assumptions

My extensions to Schlichting and Schneider's framework are driven by the needs of reconfigurable systems. Informally, I 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, I consider the class of systems with the following properties:

- The system is made up of a set of periodic applications $A = \{a_1, a_2, \dots, a_m\}$. Each a_i in A 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 A must be acyclic.

Periodicity is a common trait of dependable embedded systems [3]. Requiring specifications of functionality is appropriate in the context of dependable systems, because assuring reconfiguration correctness is secondary to assuring at least some basic level of specification correctness.

- 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 does not have to be exact; it may occur over a sparse time base [28].
- Worst-case execution times, including worst-case time to initialize data in a new configuration, can be determined for each function in a specification. This is a standard requirement of highly dependable real-time systems.
- System function can be restricted while the system is reconfigured. Time bounds on function restriction are discussed in section 4.4.
- 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 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 computers, and standard hardware error detection (and, in some cases, masking) mechanisms exist for other system elements. Commits to stable storage are atomic.

- 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 necessitates reconfiguration but involves no failure at all. This work does not provide error detection mechanisms for any system component.

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 time the application can be nonoperational; and
- A transition invariant holds during R (disallowing random state changes during R).

4.3. Reconfigurable fault-tolerant actions

4.3.1. Fault-tolerant actions

The overall timing model I 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*. Where the meaning is clear, a reconfigurable application is 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, 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 (or actions) in another application. This issue did not arise in the previous formulation of FTAs since failures were completely masked; with the possibility of reconfiguration, however, a distinction must be drawn between *application* FTAs (AFTAs) and *system* FTAs (SFTAs).

4.3.2. AFTAs

An AFTA is an action encompassing a single unit of work for an individual 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 triggered 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. In this step, all persistent application state is made consistent with what the reconfiguration mechanism must see to cause it to satisfy the precondition (starting state) of the new configuration. Each application will have a method to correct its state, such as a rollback/rollforward mechanism or a function that resets the state to some prespecified value.
- 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 (this step may occur automatically).
- State, such as gains in a control loop, is initialized to be consistent with the new specification, if needed.

4.3.3. SFTAs

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.

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 is executed after the associated action if it is interrupted. In my approach, the recovery for an SFTA during which a reconfiguration signal is generated consists of three parts: (1) each executing AFTA establishes a required postcondition and reaches a halted state; (2) each executing AFTA establishes the condition to transition to operation under the new state; and (3) each executing AFTA establishes its precondition, meaning that all state associated with the AFTA has been initialized and the application is considered to be functioning normally.

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 I introduce the System Control Reconfiguration

Analysis and Management (SCRAM) kernel. Possible configurations to which the system might move to complete its SFTA are defined by a statically-determined set of valid system transitions, and 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 component-failure signals; (2) determine the configuration to which the system should move; and (3) send configuration signals to the individual processes to cause them to respond properly to component failure.

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.

4.4. Real-time guarantees on fault-tolerant actions

In accordance with the sketched application timing model above, I informally define a reconfigurable application to have the following timing properties:

- AP1: The application responds to an external halt signal by establishing a prescribed postcondition and halting in bounded time.
- AP2: The application responds to an external reconfiguration signal by establishing the condition necessary for a transition to the prescribed configuration in bounded time.
- AP3: The application responds to an external start signal by starting operation in whatever configuration it has been assigned in bounded time.

A formal model of these properties is discussed in the next chapter. Informally, the approach I take to modeling time at the system level is to associate 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 completes with no intervening failures.

While Schlichting and Schneider's work includes a discussion of temporal properties of the system in the event of multiple successive failures, including a method to guarantee liveness, I extend the framework to cover the timing elements of reconfiguration for systems that must operate in hard real time. Depending on system requirements, 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. In the worst case, each failure cannot be dealt with until the end of the current reconfiguration cycle. 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 longest configuration sequence C_1, C_2, \dots, C_s , the maximum restriction time is $\sum_{i=1}^{s-1} T_{i, i+1}$, 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 new maximum time to successful action completion 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 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 specified amount of time (in a safe configuration, or in a configuration appropriate to all environmental conditions) before a subsequent reconfiguration takes place.

4.5. 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 low-level properties required of the architectural elements, the developer will know that the high-level properties that assure reconfiguration have been met. The specific mechanisms for showing that the architecture implies the properties are discussed in the next chapter.

4.5.1. Application reconfiguration

A system built with the candidate architecture is composed primarily 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 [38]. Each specification for each application is defined by domain experts. The experts also define system *configurations* that provide acceptable services and document the configurations in a *reconfiguration specification* [51]. System reconfiguration is the transition from one configuration to another.

The internal structure of an application is depicted in Figure 1. Each application is built of a set of modules. The modules are design components that follow decomposition rules such as would be used when implementing a non-reconfigurable system. Properties

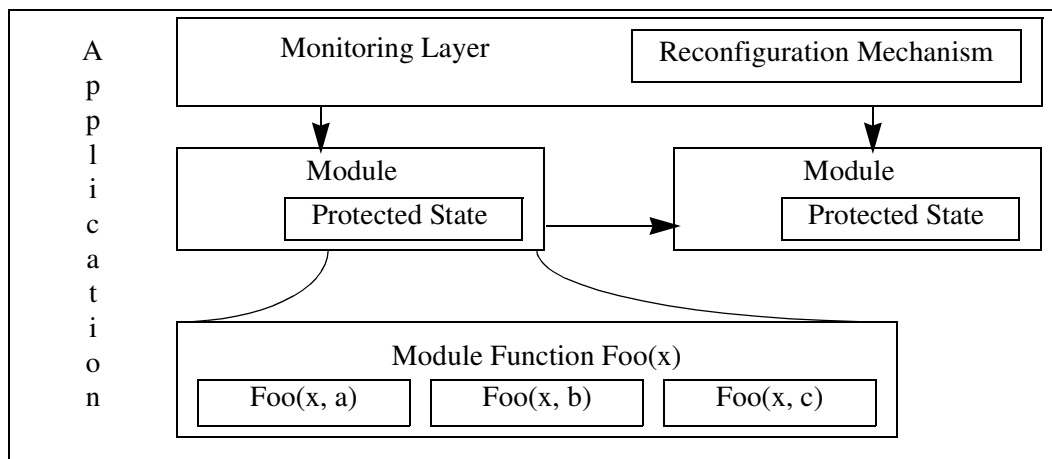


Figure 1. Application Structure

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 2.2). It is possible that some errors of interest will be detectable only at the application level, however, in which case restricting the applications' predicates to a simple conjunction of module properties is unreasonable. Detecting application-level errors that span several modules could be accomplished by creating a protection shell that combined the individual modules. The value of explicit provision for such classes of errors is unclear, however, and a general analysis of error detection and application structure to support that detection is left to future work.

Each module instantiates a standardized interface designed to support reconfiguration assurance. State property violations within a module can be detected quickly, and a reconfiguration signal can be generated if the error is not masked. Each function in a module interface presents a set of functional *service levels*. For any individual module, these services might be the same across application configurations; they must differ in

some module, however, since otherwise the application configurations will provide the same function. Application configurations are constructed from combinations of module services.

Configuring the modules to provide the correct services for the chosen 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. Second, each module's private state could include a module-specific *service level parameter*. The service level parameter instructs the interface to provide a specific type of function, ranging from basic safe service to more elaborate calculations or operations for full functionality. A call to $f_{oo}(x)$, then, would effectively become a function call to $f_{oo}(x, \text{svclvl_parm})$, where $f_{oo}(x, \text{svclvl_parm})$ is the proper version of $f_{oo}(x)$ for the module configuration corresponding to svclvl_parm . The formal model I have constructed (described in detail the next chapter) provides for either method of configuring the application. The example in Chapter 6 uses the function mapping to choose reconfiguration functions, and the service level parameter to choose application functionality.

The application's modules are linked through a *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 detected and unmasked fault during computation of a module function causes control to be returned to the monitoring layer. The monitoring layer is then responsible for interacting with the rest of the system to determine the next configuration for the application. Also, the monitoring

layer checks at the beginning of each execution cycle for any reconfiguration signals that have been sent to the application during the previous cycle, and if it sees that one has been sent, executes a slightly modified version of the functionality for the current configuration. The modifications are simple functions to save 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.

4.5.2. System reconfiguration

System reconfiguration is effected by the System Control Reconfiguration Analysis and Management (SCRAM) kernel. This kernel implements the *external* reconfiguration [38] 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. 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 monitoring layer.

Should reconfiguration become necessary, the applications 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, so that the recovery stages of all the AFTAs (described in Section 4.3.3) occur together.

The details of the forms of the various configurations, their values to the user, and the feasible target configuration from any given state are all determined before the system is

put into operation. This approach facilitates analysis of the actions that the SCRAM might take and permits stronger assurance arguments about the SCRAM itself. Also, the SCRAM kernel has a standardized interface so that applications can be easily added to or removed from the system from a reconfiguration point of view. The standardized interface enables the SCRAM to be reused across many different systems.

The temporal structure presented here 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, it 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 accounted for 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.

4.5.3. Example implementation platform

To illustrate a potential use of the informal model, I briefly describe the logical architecture together with a possible implementation platform. The combination is illustrated in Figure 2. The implementation platform includes a set of processing elements that communicate via an ultra-dependable, real-time data bus. Each processing element

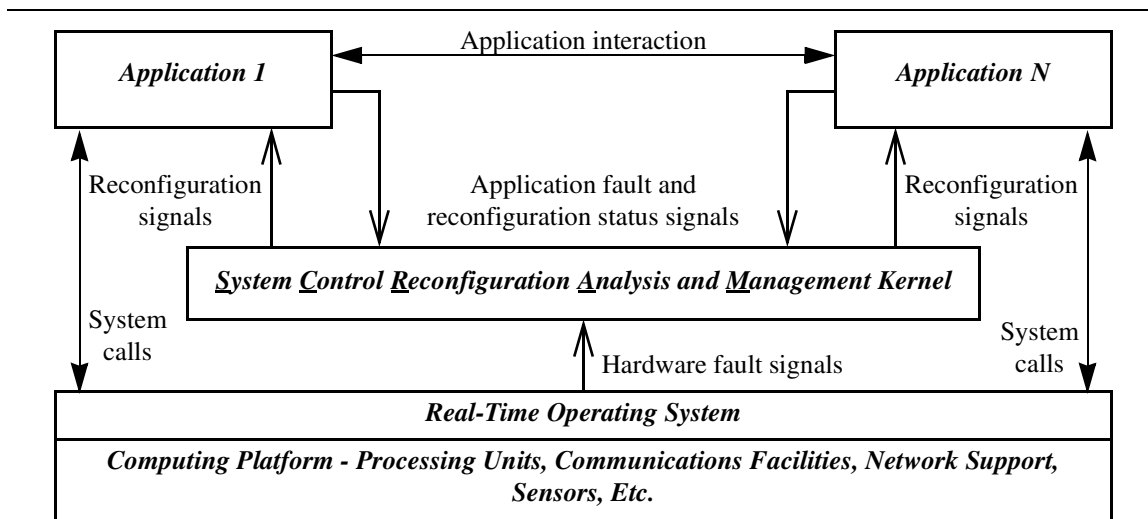


Figure 2. Logical System Architecture

consists of a fail-stop processor with associated volatile and stable storage that executes an ultradependable 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 [29]; and an example operating system might be one that complies with the ARINC 653 specification [5]. 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.

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 processor, 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

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.

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.

The above implementation platform is not the only possible platform. The formal model of the architecture (discussed in the next chapter) is a logical specification of application characteristics and interactions. An implementation of a system that has the architecture need only exhibit the formal characteristics of the architecture specification. The specifics of an argument that the system exhibits those characteristics will vary widely across different potential 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.

4.6. Summary

This chapter has presented an informal model of the overall theory of reconfiguration, and an architecture that implements that model. The model is based on Schlichting and Schneider's definition of fail-stop computing [43], where a fault-tolerant action's recovery mechanism can prepare the system for operation under a new configuration. The architecture defines composition of modules into applications, how reconfiguration is carried out within an application, and how the SCRAM carries out reconfiguration at the

system level when a system includes multiple applications. Having described a reconfigurable system intuitively, I next present the formalization of the architecture and its properties.

CHAPTER 5

Formal Model

While the discussion of reconfiguration presented in Chapter 5 provides an outline of the concepts needed to construct and assure a reconfigurable system, as an informal discussion it lacks the rigor necessary for dependability assurance. To provide this assurance, I have created an assurance argument (depicted in Figure 3) 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 I use as a 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, any instance of that architecture will

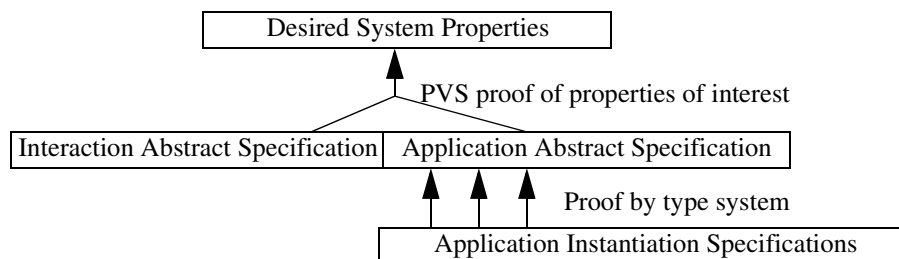


Figure 3. Reconfiguration Assurance Argument Structure

be a reconfigurable system with the stated formal properties, as long as the proof obligations generated by the type system of the formal model have been met.

My work in assurance is focused at the specification level. This level of abstraction offers many benefits in the complexity of properties that can be shown through formal analysis and in comprehensibility of what the properties mean in terms of system composition. Ensuring that an implementation matches its specification, and thus possesses the properties that are shown of the specification, is left to related work in verification and to future work in how verification can be effectively applied to reconfigurable systems.

The formal model presented in this chapter is specified in PVS, proofs of the putative theorems have been constructed, and the proofs have been checked with the PVS system. I have also formally specified the essential interfaces of an example reconfigurable system (see Chapter 6) and shown that this example has the necessary properties of the formal architecture. The result is an assurance argument based on proof. This chapter discusses the technology used to create and assure the formal model, assumptions on a system for the model to apply, the model itself, and the properties that are guaranteed to hold over the model. The complete formal specification is included as Appendix A.

5.1. The Prototype Verification System and the proof structure

To help the reader unfamiliar with the Prototype Verification System (PVS) to understand the model, I include here a brief discussion of the system. PVS has two major components: the language, and the proof-checking system.

5.1.1. The PVS language

The PVS language is a higher-order logic based on type theory. Many basic types and type manipulators are included as part of the system. User-constructed types can be either interpreted (defined as an enumeration, or in terms of other types) or uninterpreted (with no definition, and thus no implicit properties). Subtypes are defined by adding a predicate to a supertype. For a supertype $Super$, then, a subtype Sub of $Super$ with property P could be defined as $\{s: Super \mid P(s)\}$. 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.

PVS requires that functions be total (although it provides a specific type for partial functions). Therefore, the type system requires that any function's output be defined for any parameter of its input type. Also, if the function taking a parameter s of type Sub is called within another function, $P(s)$ must be true of the parameter sent to the called function as input. In some cases, these properties are undecidable. Type-checking a specification with an undecidable type system leads to the generation of type-correctness conditions (TCCs), a kind of proof obligation. In order for a specification to be considered type-correct, all of its TCCs must be discharged.

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.

5.1.2. The PVS system

The PVS system allows a developer to create specifications, state properties over those specifications, write proof scripts for the properties, 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 \top is “`true` \Rightarrow \top .” 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.

5.1.3. Reconfiguration proof structure

I 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 discharge. If the instance does not type-check, it may 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. I have created a set of functions that define rules for state traces which satisfy the specification constraints. I then proved the properties over the combination of the state trace functions and the type predicates. Essentially, the state trace functions will map to the SCRAM’s execution, with the help of bus characteristics and processor clock synchronization. This work does not address implementation of the SCRAM; platforms that can provide the characteristics needed by

the SCRAM to satisfy the state trace properties exist in practice for ultradependable systems, and so tailoring them to reconfigurable systems specifically is left for future work. The properties that the SCRAM must ensure in a system are fully specified, however. Properties for individual applications, and the system-specific data that must be input to the SCRAM, are all encoded in the type system.

The property proofs I have created are quite lengthy, since mechanical proofs must often be much more detailed than logical proofs (theorem provers don't follow leaps of intuition). 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 generated by those faults do not cause unprovable theorems to be provable. Appendix D contains an example proof script. I do not include all of the proofs in this document, although they can be found elsewhere [53].

5.2. Formalism 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 difficult to show in any rigorous way. The formal model assumes all of the system characteristics set out in Section 4.2. To enable formal specification and proof, I make the following additional assumptions:

- 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.
- Application properties of interest can be constructed through conjunction of module properties of interest, and 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 or effectively relaxed by creating an application with only one module, but I include it here to show how existing work on error detection could be incorporated into my architecture.

These assumptions impose restrictions on a system to facilitate analysis. They are not strictly necessary, but even with these assumptions the proofs are complex. Despite these restrictions, the model presented here supports the development of a useful class of systems.

5.3. Application structure

This section describes the formal models of system state, modules, and applications.

5.3.1. State

The model I have created contains a number of abstract properties guaranteed by its type system. To guarantee properties over the model'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 persistent storage. Volatile storage is not considered, since no assumptions can be made over it.

Modeling state in this way is closer to writing a specification *language* than writing a specification. PVS, for instance, has a set of rules for how an identifier can be constructed. While I use those rules to define the `data_id` type, a specifier using my framework would define instances of `data_id`, naming those instances with PVS's identifier rules.

The persistent storage of the system is represented as:

```
data_state: TYPE = [data_id -> data_value]
```

This means that in an instantiation of my framework, to create a persistent variable `var`, a specifier would write:

```
var: data_id
```

which declares some constant of type `data_id`, represented by `var`, whose value is undefined. Given a data state `st`, `st(var)` represents the value of `var` in `st`.

Using this model, it is possible that a given `data_state` could have infinitely many `data_id` \rightarrow `data_value` mappings that do not actually belong to the system. PVS possesses a theory-interpretation mechanism whereby a specifier can assign values to uninterpreted types in an imported theory. Ideally, a specifier instantiating my framework would be able to write an interpretation of the state theory that mapped the uninterpreted

`data_id` type to an enumeration type representing exactly the set of system identifiers; likewise, for `data_values`. Two issues exist with this approach. First, the `data_state` type is interpreted: it is defined over two uninterpreted types. Second, my abstract structure is constructed using several theories, and the interpretation is not imported as a general interpretation with the imported theory. Research in practical instantiation of an abstract PVS architecture is currently underway as part of future work on verification. The abstract specification makes no restrictions on data elements that do not belong to the system, so they can safely be ignored.

Because system state is modeled as a whole (rather than each application's or module's state being modeled separately, another possible solution that proved unworkable), a predicate type and a function type are provided that are restricted to part of the state. The type

```
predicate(scope: set[data_id]) : TYPE =
{p: pred[data_state] | FORALL (st: data_state) :
  (p(st) =>
    FORALL (st2: data_state) :
      (FORALL (id: (scope)) : st2(id) = st(id)) => p(st2))}
```

defines a predicate that depends only on the corresponding values of data identifiers included in `scope`. This is used to specify predicates over a particular module's state. The type

```
func(scope: set[data_id]) : TYPE =
[# pre: predicate(scope),
  f: {f: [data_state -> data_state] |
    FORALL (d: data_state, id: data_id) :
      NOT scope(id) => f(d)(id) = d(id)}
#]
```

defines a function that can change only the values of data identifiers included in `scope`, but whose result can depend on any element of the state. `pre` specifies the precondition for the function. Because PVS requires total functions, a function's precondition generally equates to the type predicates over its inputs. Because of the type composition used in this architecture, restricting function input types so that the function's output will meet its postcondition over the entire input space is likely to be impractical for many system instantiations. Also, the types of the functions must be the same for efficient manipulation of the universal quantifiers used in the proofs. Methods for refining specification instances to facilitate verification against a high-level language are discussed in Chapter 7 as part of future work.

5.3.2. 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. 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. The formal elements of a module are:

<code>scope:</code>	The persistent data elements that can be manipulated by the module.
<code>sv:</code>	The service levels the module can provide.
<code>svlvl_parm:</code>	The data element that holds the current value of the module's service level parameter.
<code>inv:</code>	The module invariant for each service level.
<code>pre, trans, post:</code>	The module's precondition for correct operation under each service level, its transition condition for beginning initialization under each

service level, and the postcondition that must be established when a reconfiguration is signaled under each service level, respectively. These conditions must subsume the invariant for the same service level.

5.3.3. Applications

An application is made up of a set of modules that provide some coherent system-level functionality. An application provides guarantees over its state composed from the guarantees provided by its constituent modules. The functions that are called to effect reconfiguration are defined at the application level to change modules' internal state in a coordinated way. The formal elements of an application are:

<code>svcs:</code>	Possible application configurations.
<code>modules:</code>	The modules that comprise the application. The data identifiers in the modules' respective scopes must be disjoint.
<code>svcmmap:</code>	The mapping from application configurations to module service levels.
<code>execute:</code>	Entry point to application function (“ <code>main()</code> ”) for each configuration. Its postcondition must subsume all module invariants for that configuration.
<code>exec_halt:</code>	Entry point to application function (“ <code>main()</code> ”) for each configuration, to be called when a possible reconfiguration has been signaled. Its postcondition must subsume all module postconditions for that configuration.
<code>halt:</code>	Function to prepare an application to reconfigure, if the application has signaled a reconfiguration (and thus might not be able to complete

`exec_halt`). Its postcondition must subsume all module postconditions for that configuration.

`prep`: Function for the application to meet its transition condition. Its precondition is implied by the conjunction of module postconditions for any configuration, and its postcondition must subsume the conjunction of module transition conditions for the new configuration.

5.4. System model and the SCRAM

The details of application interaction at the system level during a reconfiguration are captured in the *reconfiguration specification*. It defines relevant characteristics of the operating environment, system configurations, and transition information. It contains all of the system-specific information that must be provided to the SCRAM to ensure that appropriate reconfigurations occur when necessary.

5.4.1. 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. The details of the operating environment that affect the relative utility of system configurations need to be identified so that if a reconfiguration is signaled, a new configuration appropriate to the circumstances can be chosen.

In the formal model, the environment is modeled by a rudimentary type system, consisting of a set of `env_ids` (environmental characteristics) and `env_params` (values

that the characteristics can take). The level of abstraction at which the specification is defined makes the specification of separate types for different environmental factors inconvenient, so instead I use the structure `valid_env` to restrict which values are possible for each characteristic. The type `env(v: valid_env)` is, in essence, parameterized by a specific environmental type system, and represents a particular mapping from each `env_id` in that type system to some `env_param` that is a type-correct instance of the `env_id`.

Some members of the type `env(v: valid_env)` for a particular `valid_env` could be unreachable. For instance, if an actuator is supplied with power from a particular power source, and the status of both the actuator and the power source were represented as environmental characteristics, the state `{actuator = working, power source = failed}` can never be reached. It is important to know this, because there is no need to specify a system transition to cover this case. Without a specific listing of reachable or unreachable states, there is no way to define a transition coverage test except one that requires all transitions be covered, even if those transitions will never happen in practice.

Some transitions also might not be possible; for instance, repair might not be possible for an actuator during aircraft flight, in which case the actuator will never return to a working state after it has failed. Again, the system need not prepare for this possibility, but must know which cases it has to cover. The `env_txn` type encodes possible environmental state changes given a particular set of reachable states, and the `reachable_env` type encodes all reachable states and possible transitions for a given system.

To represent a particular system's environment, the reconfiguration specification contains:

- E:** an instance of type `valid_env` (i.e., an encoding of the operating environment's type system); and
- R:** an instance of type `reachable_env` (i.e., an encoding of the possible environmental transitions, encoded in the type system of \mathbb{E}).

5.4.2. System configurations

A reconfiguration specification also contains the applications that make up the system, including the applications' configurations, and information on overall system configurations and transitions:

- apps:** The set of applications in the system. All module scopes in the system must be disjoint. (Communication takes place by reading `data_ids` in other modules' scopes.)
- app_seq:** The sequence in which applications must execute in order for dependencies to be preserved. In practice, applications will execute in parallel, with synchronization primitives used to preserve the dependencies. Because the formal model is written in a declarative specification language, correct behavior is defined as any behavior that leaves the system in the same state as that which would have resulted from execution of the applications in the stated order.
- s:** Labels for system configurations. Actual configurations are defined in the SCRAM table (described below).
- choose:** Function from combinations of configurations and environmental states to new configurations. This information is provided in the top-level

specification because it should be included in inspections at the requirements level.

`T`: The time allowable to transition from any specification to any other specification. The type for `T` requires that `T` be ≥ 4 execution cycles, which is how I ensure enough time is available to transition. Generally, `cycle_time` (the time for one periodic execution cycle) will be very small, so this requirement is reasonable.

5.4.3. System transitions and the SCRAM

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 data elements of a system transition:

`source`: Source configuration. `s` is the set of possible system configurations, as stated above; `source` must be of type `(S)`, which indicates that `source` is a member of the subtype of configurations for a particular system, rather than an arbitrary configuration.

`target`: Target configuration.

`trigger`: Environmental state in which the transition is appropriate. This is restricted to possible environmental states of the system.

The system-specific state passed to the SCRAM is defined in the `SCRAM_table` type. Its elements are:

`configs`: The mapping from system configuration labels to application states that comprise the configuration.

`primary`: The configuration that the system is in initially.

<code>safe:</code>	The set of configurations for which a reconfiguration signal may not occur during operation. This set enables the maximum time to an operational system to be characterized.
<code>start_env:</code>	Possible initial environments for the system (e.g., all environments in which all system hardware components are functional).
<code>txns:</code>	The set of transitions the system can take. The <code>covering_txns</code> predicate must hold over all elements of <code>txns</code> ; it 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 reconfiguration specification also contains an element of type `SCRAM_table`.

5.5. State traces

The drawback of declarative specifications is that the specifier can only restrict *what is observed* during system operation; the specification does not tell a system to *do* anything. Temporal logics are better suited to expressions of properties over time. An initial formalization of this work [50] was in Real-Time Logic (RTL) [24], but RTL is not powerful enough to express the predicates over state that a reconfigurable system must meet. The PVS model developed as part of this work 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. As explained earlier, in order to specify state-change restrictions over time, the model defines possible system traces that can be legally generated by a system which complies with a particular reconfiguration

specification. This is a common technique in temporal specification with non-temporal languages.

The assumptions on synchrony and equal execution times were made for simplicity in specifying state traces (and proving properties over those traces). Thus, a reconfiguration either: (1) takes 4 cycles, including the signal generation cycle, to complete successfully; (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 signal is generated and serviced before the 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, application i has generated a reconfiguration signal, all applications will reconfigure, and the new system configuration will be C_t .

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) \rightarrow all apps	Applications prepare to transition to C_t	Application transition conditions for C_t
4 (end)	SCRAM: initialize \rightarrow all apps	Applications initialize, establish operating state for C_t	Application preconditions for C_t

5.5.1. System state

A reconfigurable system has a number of elements of system state that do not belong to any application (or do not exist as explicit state—e.g., time). These elements, collected in the `sys_state` type, are:

<code>sp:</code>	The reconfiguration specification associated with the state. (This must be contained as part of the state to enable parameterized type instantiation.)
<code>st:</code>	The state that the system is manipulating explicitly (i.e., contents of applications' stable storage).
<code>reconf_st:</code>	The reconfiguration stage of each application.
<code>app_svclvls:</code>	The configuration of each application.
<code>app_last_svcs:</code>	The previous configuration of each application (to be precise, the configuration of each application during the last SFTA).
<code>svclvl:</code>	The configuration identifier for current system state. This is represented separately from the application configurations because if the system enters an inconsistent state, one cannot be inferred from the other.
<code>last_svc:</code>	Likewise, the previous configuration identifier.
<code>t:</code>	The time at which the state is true of the system.

5.5.2. Application execution

Application execution is modeled by specifying the actions of the monitoring layer for an application. Recall that the monitoring layer is the element with overall control at the application level. It chooses each cycle whether to carry out an application's 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. The function representing the monitoring layer, then, is:

```
monitor(st: sys_state, app: (st`sp`apps)) : sys_state =
  CASES st`reconf_st(app) OF
    normal      : execute(st, app),
    interrupted : st,
    halting     : halt(st, app),
    exec_halting : exec_halt(st, app),
    prepping    : prep(st, app),
    training    : execute(st, app)
  ENDCASES
```

Function execution of all applications must behave in a manner equivalent to calling them in sequence over the system state. This is done in the `recursive_monitor` function:

```
recursive_monitor(st: sys_state, apps: finseq[(st`sp`apps)],
  n: below[apps`length]) :
  RECURSIVE sys_state =
    IF (n = 0) THEN monitor(st, apps(0))
    ELSE monitor(recursive_monitor(st, apps, n-1), apps(n))
    ENDIF
  MEASURE n
```

Execution at the system level is carried out by the `system_monitor` function. It 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), and if no other signal has been generated (a signal occurring at this stage results in the only case in which the reconfiguration status of the applications might not be synchronized), the `system_monitor` function 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 `recursive_monitor` function is called, and 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 (using the `next_config` function), and then the `recursive_monitor` function is called to carry out application execution for the current cycle.

```

system_monitor(st: sys_state, e: env(st`sp`E)) : sys_state =
  IF ((EXISTS (app: (st`sp`apps)) :
    st`reconf_st(app) = halting) AND
    (NOT EXISTS (app: (st`sp`apps)) :
    st`reconf_st(app) = interrupted))
  THEN LET (next_svc: (st`sp`S)) =
    st`sp`choose(st`svclvl, e) IN
    recursive_monitor(st WITH [
      `reconf_st :=
        (LAMBDA (app: (st`sp`apps)) :
          IF st`sp`SCRAM_info`configs(next_svc)(app) /=
            st`sp`SCRAM_info`configs(st`svclvl)(app)
          THEN prepping
          ELSE normal ENDIF),
      `app_svclvls :=
        (LAMBDA (app: (st`sp`apps)) :
          st`sp`SCRAM_info`configs(next_svc)(app)),
      `app_last_svcs :=
        (LAMBDA (app: (st`sp`apps)) : st`app_svclvls(app)),
      `last_svc := st`svclvl,
      `svclvl :=
        next_svc], st`sp`app_seq, card(st`sp`apps)-1)
  ELSE recursive_monitor(next_config(st), st`sp`app_seq,
    card(st`sp`apps)-1)
  ENDIF

```

5.5.3. Environmental transitions

The final element needed to construct a state trace specification is a function to define valid sequences of environmental states. The `valid_env_trace` function restricts state traces of the environment to be sequences where any two consecutive environmental

states in the sequence: (1) are the same environmental state; or (2) map to a valid environmental transition.

5.5.4. State trace

With these pieces in place, it is now possible to define a valid sequence of system states. A valid sequence 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 function application of `system_monitor` to the previous state; and (4) the system state is synchronized with the environment. The `sys_trace` type formalizes these requirements.

```
sys_trace : TYPE =
[# sp: reconf_spec,
  env: valid_env_trace(sp`E, sp`R),
  tr: {c: [cycle -> {s: sys_state | s`sp = sp}] |
    (FORALL (c1, c2 : cycle) :
      c1 + 1 = c2 => c(c2) =
        system_monitor(c(c1), env(c1 * cycle_time))) AND
    (FORALL (app: (sp`apps)) :
      c(0)`reconf_st(app) = normal) AND
    inv(sp, c(0)`svclvl, c(0)`st) AND
    (FORALL (cycl: cycle) :
      EXISTS (cyc2: cycle) : cyc2 > cycl AND
      FORALL (app: (sp`apps)) :
        c(cyc2)`reconf_st(app) = normal)}
#]
```

5.6. Reconfiguration definition

Having presented the above model, I now discuss the high-level properties I require of any reconfiguration. Defining them in an abstract sense allows me to argue that the general requirements needed for assured reconfiguration have been met. The model was constructed and refined to enable proof of these properties.

I begin by defining reconfiguration informally as:

the operation through 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 potentially 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, I characterize assured reconfiguration as certain properties that must hold over any sequence of states whose reconfiguration status is not `normal`. The sequence of states is represented by two natural numbers, representing the beginning and ending system execution cycle for that reconfiguration:

```
reconfiguration: TYPE = [# start_c: cycle, end_c: cycle #]
```

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_reconfs` function.

The following properties are those that I use to define reconfiguration.

CP1: THEOREM

```
FORALL (s: sys_trace, r: (get_reconfs(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))
```

This property 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: 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))
```

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.

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)
```

CP3 requires that all reconfigurations complete within their required time bound. As explained above, proof of this property is straightforward because of the synchrony of the system model, but it is still important to represent at a high level of abstraction.

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))

```

As the reconfiguration architecture has been written and refined, I have found a significant number of flaws when I 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. 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: 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`svcmmap(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`svcmmap(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`svcmmap
            (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

```

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.

5.7. Architecture instantiations

As explained above, any system reconfiguration specification that complies with the architecture is guaranteed to possess the high-level properties listed in Section 5.6. This

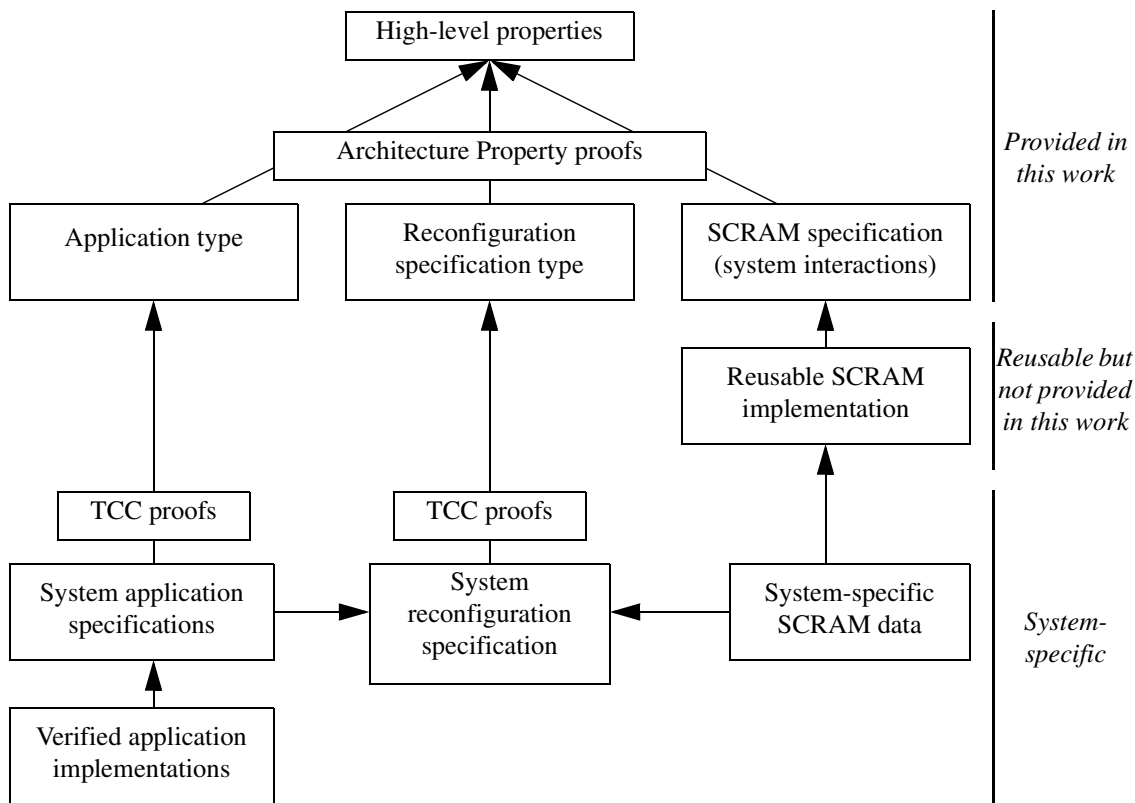


Figure 4. Software Products for a Specific System

section sketches the development process for a reconfigurable system built with the architecture. The products of this process are illustrated in Figure 4. Chapter 6 presents an example system specification that complies with the architecture.

To use the architecture for a particular system, the system's designers would first specify each application in the system as an instance of the PVS type for applications (described in Section 5.3.3). They would then determine configuration, environment, and transition characteristics of the system that must be provided to the SCRAM. They would combine those characteristics with the application specifications to create a specification of the system as an instance of the PVS reconfiguration specification type (described in Sections 5.4.2 and 5.4.3).

The system developers would then need to ensure that their reconfiguration specification instance had the properties required by the architecture. To do this, they would type-check the reconfiguration specification instance against the abstract type definition for a reconfiguration specification. Given the command to type-check the specification, PVS would either decide that the specification complied with the architecture, or would generate proof obligations. The developers would discharge the proof obligations to ensure that the system specification had the high-level properties from Section 5.6.

Once the specification instance had been constructed and verified against the architecture, an implementation of the system would be built. Section 4.5.3 describes a possible implementation platform. A high-level language implementation of each application would need to be constructed. The SCRAM is a reusable component; future work in practical application of reconfiguration includes constructing a SCRAM implementation that is parameterized over system-specific information. Given such an implementation, the designers would provide the configuration, environment, and transition parameters in the reconfiguration specification to the standard SCRAM implementation. The designers would verify that the implementation satisfied the specification (possibly with the help of safe programming, safety kernels, or protection shells). Finally, they would conduct system tests to check that the formalisms used to model the system matched the desired system function.¹

1. The extent of the system tests would be defined by the appropriate regulatory agency. The Federal Aviation Administration, for instance, currently allows primary function to be tested using the requirements for one criticality level lower than the defined system criticality level when a backup function exists.

5.8. Summary

I have constructed a formal framework in PVS, defined properties I want to be true of the framework, and proved the properties using PVS. The specification is a set of types that enforces all of the desired system properties by placing type restrictions on any instantiation. An instantiation of the architecture defined in PVS would be a set of PVS specifications for the applications in the system, along with system-specific configuration and environmental information.

The 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 type system. Because of the structure of the formal model within PVS, compliance with the type system automatically implies compliance with the high-level properties. Thus, the proofs of the high-level properties can be used with no modifications for any system that meets the type requirements. An example of an instance of the architecture is presented in the next chapter.

CHAPTER 6

UAV Example

6.1. Introduction

In order to assess the feasibility of using the approach outlined in this work and to demonstrate the concepts that constitute the approach, I have specified an example reconfigurable system. The full specification can be found in Appendix B. The system 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 functional applications:

Sensors

The `sensors` application generates simulated values of altitude and heading that would normally be read from the aircraft's environment. This application also monitors environmental characteristics (described below) that affect the appropriateness of a particular system configuration for the current environment.

FCS

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.

Autopilot

The `autopilot` can be programmed with a target altitude or a target heading, and will send commands to the FCS based on the aircraft's deviation from the target.

Pilot interface

The `pilot_interface` application simulates pilot commands and transmits the commands 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 implements 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. The system switches independently of the reconfigurable system if one of its components fails; it merely provides details of its state to the system. Failure of the aircraft's alternator causes the aircraft to switch to its backup battery power source, and at that point some computations need to be 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

Two types of unmasked faults are possible in the autopilot: a failure of the heading control subsystem, and a failure of the entire system. The specifics of the originating faults are not specified. While the failure comes from within one of the functional 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).

6.2. System configurations

My 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. Thus, they are assumed to be working correctly in each configuration. Other applications can be in different configurations, according to the possible failures described above. The system configurations are shown in Table 1 (the sensors and pilot interface configurations are not listed because they are the same in all system configurations, as described below).

6.3. Environmental states and transitions

There are three relevant environmental factors in the system:

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

Table 1: System Configurations

Electrical power generation system (electrics)

Power can be either generated by the alternator (alternator) or supplied by the battery (battery).

Rudder status (rudder)

The rudder can be working properly (working), stuck hard-over to the left (hard_over_left), or stuck hard-over to the right (hard_over_right).

Autopilot status (autopilot)

The autopilot is able to provide all services (`fullsvc`), able to provide altitude hold only (`alt_hold_only`), or not able to function (`disabled`).

Transitions are limited to only those that degrade aircraft capabilities. No assumptions are made about coincident failures.

6.4. 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. Any transition that satisfies these criteria is included.

6.5. Applications

The modular structure of the system applications is as follows:

Sensors and Pilot Interface

The `sensors` application and the `pilot_interface` application are assumed not to fail, and so have only one configuration each. Also, because they represent rather than provide functionality in this system, they include only one simple module.

Autopilot

The autopilot has only one module, but this module has three service levels: `ap_compute_standard`, `ap_compute_ah_only`, and `ap_compute_off`. `ap_compute_standard` executes the full autopilot function, including both heading and

altitude hold capabilities. `ap_compute_ah_only` executes altitude hold function only. `ap_compute_off` disables the autopilot.

Flight Control System (FCS)

The FCS application has three modules. The first module, `FCS_calc`, 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 module, `FCS_adjust`, modifies the output of `FCS_calc` to compensate for a rudder hard-over condition, if one exists. The third module, `FCS_output`, transmits the (modified or unmodified) output of `FCS_calc` to the actuators. This third module allows separation of concerns to be maintained. `FCS_adjust` does not modify the values in the scope of `FCS_calc` directly; `FCS_calc` and `FCS_adjust` have their own state, and `FCS_output` chooses the appropriate values to transmit to the acutators.

The reconfiguration interfaces for the four applications described above, the nine acceptable configurations, and the transitions between configurations are specified in PVS. The instantiation has been type-checked against the abstract specification described in Chapter 5, and the generated proof obligations have been discharged. The proof obligations for the flight control application and the example reconfiguration specification are documented in Appendix C.

6.6. An example system fault-tolerant action

In the example instantiation, each AFTA and each SFTA execute as described in Sections 4.3 and 5.5. 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

Frame	Stage	Action	Predicate
1 (start)	Sensors: interrupted All other applications: normal	Sensors: signal generated All other applications: normal execution	Sensors: invariant All other applications: invariant
2	Sensors: halting All other applications: exec_halting	Applications anticipate possible reconfiguration	Application postcondi- tions
3	SCRAM: prepare(C_i) \rightarrow all apps	FCS: prepare to compensate for rudder failure All other applications: normal execution	FCS: transition condition All other applications: invariant
4 (end)	All applications: normal	All applications: normal execution	All applications: invariant

Table 2: Example Reconfiguration Stages

reconfiguration steps for this case is shown in Table 2. 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; the `sensors` application executes its `halt` function, while the other applications execute their `exec_halt` functions during this frame. 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. My 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.

6.7. Compliance properties

As explained in the last chapter, 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 discharged with a single command, since the conditions included in the example are relatively simple. The TCC requiring the most complex proof to discharge is shown in its entirety in Figure 1. Manipulated for illustration, the obligation becomes:

```

prototype_reconf_spec_TCC2: OBLIGATION
  FORALL (x: speclvl):
    proto_speclvl(x) IFF
      % x does not map to the specification label that
      % represents an inconsistent configuration and
      proto_speclvl(x)
  AND % essentially a repeat of the previous requirement
  AND covering_txns(proto_apps,
                    proto_speclvl,
                    proto_valid_env, proto_reachable_env,
                    proto_SCRAM_table`txns, proto_SCRAM_table`primary,
                    proto_SCRAM_table`start_env)
  AND % repeat of the first requirement
  AND proto_speclvl(proto_SCRAM_table`primary)
  AND % repeat of the first requirement

```

```

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

```

Figure 1. TCC from the Example System

The only nontrivial conjunct in this proof obligation is the `covering_txns` requirement. As described in the last chapter, 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 not difficult but was time-consuming. Default PVS proof strategies do not easily find the appropriate instantiation for a particular goal amongst a number of concrete representations. This difficulty could be easily overcome with a simple enumeration tool (or, perhaps, PVS strategy) to choose appropriate instantiations. In general, assurance of a reconfigurable system requires a detailed analysis of system execution under all possible failure scenarios, and so formalizing this requirement has not added an analysis burden to the developer. Formalizing the necessary analysis is likely to make system design easier rather than otherwise.

6.8. Implementation

An implementation of an earlier version of this example [52] exists, although the implementation has not been verified since the emphasis of my 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 [14].

6.9. Summary

This chapter has described an example instantiation of the formal reconfiguration framework. The example system has four applications, five possible failure modes, and nine configurations. The PVS system specification has been type-checked against the abstract framework, and all proof obligations have been discharged. This means that the example has the abstract reconfiguration properties described in the last chapter. An earlier version of the example has been implemented in Java and is able to reconfigure successfully.

CHAPTER 7

Conclusion, Contributions, and Future Work

This chapter presents the contributions of the work. Section 7.1 summarizes the work and its impact. Section 7.2 lists its specific contributions. Section 7.3 describes the research program my colleagues and I have developed that I anticipate will increase the impact of my work on software engineering practice.

7.1. 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

would have to be assured, reducing the complexity and cost of software analysis in many systems. At the hardware level, fail-stop machines 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 it is critical to meet functional and timing constraints during reconfiguration as well as during standard operation. This work has demonstrated a means through which general properties of reconfiguration can be assured via proof for real-time, periodic critical systems.

Existing approaches to achieving dependability through reconfiguration involve building the main system and then adding the capability to transition to a separate backup. My work advocates 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 infrastructure for applying it has the potential to change the way designers think about critical software. By distinguishing between desirable function and necessary function, they can build systems with significantly higher assurance of dependability, while retaining complex function 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, my 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. This work sets out an assurance argument at the system level consisting of: (1) assurance of noninterference of noncrucial function with crucial function; (2) assurance of crucial function; and (3) assurance that reconfiguration will proceed correctly.

7.2. Contributions

The claims made in the preceding section are supported by the following specific contributions:

Definition of reconfiguration

The intuitive definition of reconfiguration is relatively clear. Intuition is not sufficient for assurance, however. This work has defined reconfiguration as a set of properties, so that assured reconfiguration has a precise meaning that is characterized formally. The

properties can apply to a wide range of periodic systems. The specific assumptions on applicable systems are stated.

Reconfiguration architecture

In this work, I have created an architecture within which system software applications can be placed. The modular structure within an application supports the use of current error detection techniques within the architecture. The formal representation of the architecture takes the form of a type system, where architecture properties are defined as type constraints, so that architecture compliance reduces to type compliance.

Proofs of architecture properties

The software architecture I have created is general enough to be reusable across a number of systems. I have shown that overall assurance properties of my architecture hold, and so designers do not have to show that those properties hold for each system that employs the architecture. Thus, assurance of overall reconfiguration reduces to assurance of compliance with the architecture.

Automatic generation of application-specific compliance requirements

Because architecture compliance, and thus reconfiguration assurance, reduces to type compliance, PVS can be used to automatically generate application-specific proof obligations for reconfiguration assurance in a specific system. The obligations are generated within the theorem-proving system, so that a developer can discharge them using PVS.

A general mechanism for reconfiguration assurance

While my framework is designed to enable reconfiguration triggered by an error, reconfiguration triggers do not have to be errors. The framework I have built can be used to make reconfigurable systems in general more dependable because it separates the different aspects of operation and provides guarantees on their composition.

A strategy for proof reuse

While previous work on proof reuse exists, it focuses on providing increasingly more abstract constructs from which a proof can be built. My work reasons about system properties at a very abstract level, so that the proof reuse is over system composition rather than layered formalisms. It can be combined with layered formalisms to reduce the conceptual distance from logic gates on hardware to abstract properties of application interaction.

7.3. Future work

While building and assuring a reconfigurable system with the architecture I have provided is theoretically possible, some practical barriers to building production reconfigurable systems still need to be addressed. For the most part, these barriers are not specific to reconfiguration. They are also needed when applying formal analysis of the kind described in this work to non-reconfigurable systems. My colleagues and I have begun to solve some of the remaining problems, while I plan to address the others in the future.

Error detection

Reconfiguration in response to software errors is only as good as the error detection mechanisms available for a system—one can do nothing about errors that are unidentifiable. While a broad range of research results in the field of error detection exists, a comprehensive characterization of error states and a coordinated theory of how to detect them will significantly strengthen the applicability of my results. Research on this is currently underway.

Verification

Formal verification of an implementation against its specification can be done in theory using Floyd-Hoare analysis. Such analysis can be extremely tedious, however, particularly when the specification is very abstract (as PVS specifications often can be). Automatic code generation is gaining increasing prominence under the name *model-based development*, but its success is primarily confined to control systems. A general verification mechanism would significantly increase practicality of the approach.

My colleagues and I are researching a technique for verification that closely models the Floyd-Hoare pattern, but at a more abstract level. The overall technique is depicted in Figure 5. The specific strategy is to manually translate a PVS specification into code written in the SPARK Ada subset along with appropriate SPARK annotations [9]. The SPARK Examiner, a static analyzer, ensures that the code complies with the annotation properties. The annotations are then automatically translated into a PVS specification. Finally, equivalence of the new PVS specification to the original PVS specification is shown by proving a theorem of equivalence in PVS. For the SCRAM's distributed

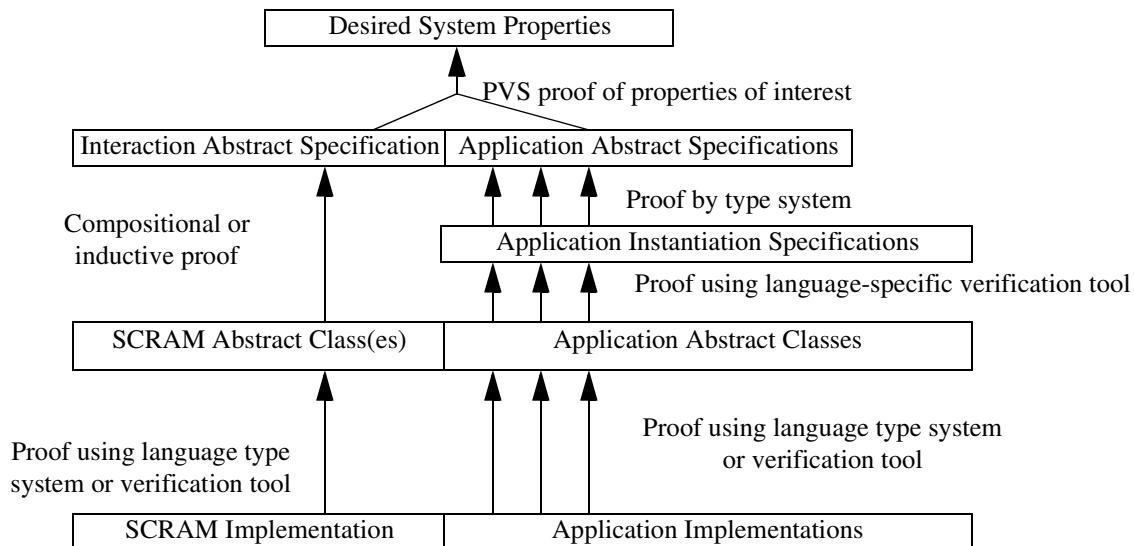


Figure 5. Reconfiguration Assurance Argument Structure

properties, we plan to use an inductive proof over execution cycles, based on SPARK annotations in the SCRAM code.

We believe that structuring the manual translation, and recording some of the translation details for a particular specification, can inform the automatic specification generation process to an extent that proving the equivalence of the two PVS specifications is possible in practice. A first iteration of this approach, based on part of the example presented in Chapter 6, has been completed.

Patterns for verification within the architecture

While working on the general verification problem, my colleagues and I realized that providing a method to explicitly separate application functionality from its reconfiguration interface would be very helpful. Explicit separation would allow developers to distinguish critical function more clearly and allow them to specify noncritical function without using PVS. This would also help ease the burden of

idiosyncrasies of the current reconfiguration architecture, and facilitate development of tool support for automatic generation of TCC proofs.

Integration with architecture formalisms

The disadvantage of creating an architecture to achieve a specific goal is that systems generally can have only one architecture. Therefore, it would be very helpful to have a method of integrating the architecture specifying reconfiguration function with current architecture formalisms. Furthermore, combining the software architecture with the reconfiguration architecture will show more clearly what interactions must occur between applications during reconfiguration. This might permit some relaxation of the requirements imposed by the architecture, for instance, the requirement on acyclic dependencies.

Model checking of properties

Verification is not the only way to assure compliance with a specification. Model checking can not only guarantee a number of desired system properties (particularly in concurrent systems); it has also been shown to be practical in industrial use. My colleagues and I, along with colleagues at Politecnico di Milano, plan to research how model checking can be used to verify implementation properties derived from specification properties stated axiomatically using TRIO [19], Politecnico di Milano's real-time specification language. This will help define what is the most useful intersection of verification and model checking for a reconfigurable system: some properties might be easier to verify, and others to model check. Constructing a framework to support both would be of great advantage to

developers in practice, who have to choose between a wide range of alternatives when assuring their systems.

Situated formalisms for reconfigurable systems

While my work is aimed at managing complexity in systems so that humans can reason about critical function more effectively, it still does not address the basic question of how the system being built relates to the environment in which it will operate. The only way to do this is to support the process of formalization from a system designer's informal perception of what the system should do. I have previously conducted research in supporting software validation by the systematic introduction of natural language into a software product (e.g., specification or source code) [49]. In addition to general further research on this topic, I plan to explore what aspects of reconfiguration are most problematic to understand, what informal concepts are unique to reconfiguration, and how formalization of those concepts can be supported.

Bibliography

- [1] See http://www.airbus.com/product/a320_flight_deck.asp "Airbus - Aircraft Families - Passenger Comfort," accessed April 2005.
- [2] Allen, R., R. Douence, and D. Garlan. "Specifying and Analyzing Dynamic Software Architectures." Proc. 1998 Conference on Fundamental Approaches to Software Engineering (FASE '98) Lisbon, Portugal, March 1998.
- [3] 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.
- [4] Anderson, T., and R. W. Witty. "Safe programming." *BIT* 18:1-8, 1978.
- [5] ARINC Inc. "Avionics Application Software Standard Interface." ARINC Specification 653, Baltimore, MD, 1997.
- [6] Avizienis, A. "The N-version approach to fault tolerant software." *IEEE Transactions on Software Engineering* 11(12):1491-1501, December 1985.
- [7] Avizienis, A., J.-C. Laprie, B. Randell, and C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing." *IEEE Transactions on Dependable and Secure Computing*, 1(1):11-33, January 2004.
- [8] 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.
- [9] Barnes, J. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, London, 2003.
- [10] Bradbury, J. S. "Organizing Definitions and Formalisms for Dynamic Software Architectures." Technical Report 2004-477, Queen's University, Kingston, Ontario, Canada, March 2004.
- [11] Brilliant, S.S., J.C. Knight, and P.E. Ammann. "On The Performance of Software Testing Using Multiple Versions." Twentieth Annual Symposium on Fault-Tolerant Computing, June 1990, Newcastle-upon-Tyne, England.
- [12] Budhiraja, N., K. Marzullo, F. B. Schneider, and S. Toueg. "Optimal Primary-Backup Protocols." Workshop on Distributed Algorithms, Haifa, Israel, November 1992.
- [13] Burns, A., and A. J. Wellings. "Safety Kernels: Specification and Implementation." *High Integrity Systems* 1(3):287-300, 1995.

- [14] 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.
- [15] 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.
- [16] Collinson, R. P. G. *Introduction to Avionics Systems*. 2nd ed. Kluwer Academic Publishers : Boston, 2003.
- [17] See <http://www.dependability.org/wg10.4/> "IFIP WG10.4 on Dependable Computing and Fault Tolerance," accessed April 2005.
- [18] Federal Aviation Administration Advisory Circular 25.1309-1A, "System Design and Analysis."
- [19] Furia, C. A., D. Mandrioli, A. Morzenti, M. Pradella, M. Rossi, P. San Pietro. "Higher Order TRIO." Internal Report 2004.28, Dipartimento di Elettronica ed Informazione, Politecnico di Milano, September 2004.
- [20] 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.
- [21] Hayhurst, K. J., D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson. "A Practical Tutorial on Modified Condition/Decision Coverage." NASA/TM-2001-210876, NASA Langley Research Center, Hampton, Virginia, May 2001.
- [22] Horning J. J., H. C. Lauer, P. M. Melliar-Smith, and B. Randell. "A program structure for error detection and recovery." Symposium on Operating Systems 1974: 171-187. See http://www.cs.ncl.ac.uk/old/events/anniversaries/40th/webbook/dependability/prog_str/prog_str.html.
- [23] Howden, W. E. "Reliability of the Path Analysis Testing Strategy." *IEEE Transactions on Software Engineering* 2(3):208-215, 1976.
- [24] Jahanian, F., and A.K. Mok. "Safety Analysis of Timing Properties in Real-Time Systems." *IEEE Trans. on Software Engineering*, 12(9):890-904.
- [25] Knight, J. C., and N. G. Leveson. "An Experimental Evaluation of the Assumption of Independence in Multi-version Programming." *IEEE Transactions on Software Engineering* 12(1):96-109, January 1986.
- [26] 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.
- [27] Knight, J. C., E. A. Strunk and K. J. Sullivan. "Towards a Rigorous Definition of Information System Survivability." DISCEX 2003, Washington, DC, April 2003.
- [28] Kopetz, H. "Time-Triggered Real-Time Computing." IFAC World Congress, Barcelona, Spain, July 2002.
- [29] Kopetz, H, and G. Grunsteidl, "TTP-A For Fault-Tolerant, Real-Time Systems", IEEE Computer, Volume 27, Number 1, January 1994.

- [30] Leveson, N., T. Shimeall, J. Stolzy and J. Thomas. "Design for Safe Software." AIAA Space Sciences Meeting, Reno, Nevada, 1983.
- [31] Lutz, R. R. "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems." Proceedings of the IEEE International Symposium on Requirements Engineering. IEEE Computer Society Press: Los Alamitos, CA, January 1993.
- [32] 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 (DCCA '99), San Jose, California, Jan. 1999.
- [33] Myers, J.F. "On Evaluating The Performability Of Degradable Computing Systems." *IEEE Transactions on Computers* 29(8):720-731, August 1980.
- [34] Myers, J.F., and W.H. Sanders. "Specification And Construction Of Performability Models." *Proc. Second International Workshop on Performability Modeling of Computer and Communication Systems*, Mont Saint-Michel, France, June 1993.
- [35] Neema S., T. Bapty, and J. Scott. "Adaptive Computing and Run-time Reconfiguration." Proc. Military Applications of Programmable Logic Devices, Laurel, MD, September 1999.
- [36] Peters, D. K., and D. L. Parnas. "Requirements-based monitors for real-time systems." *IEEE Trans. on Software Engineering*, 28(2), Feb. 2002.
- [37] Perrow, C. *Normal Accidents: Living with High-Risk Technologies*. Princeton University Press: Princeton, New Jersey, 1999.
- [38] 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.
- [39] Potter, B., J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Prentice Hall: London, 1996.
- [40] RTCA. "Software Considerations in Airborne Systems and Equipment Certification," document RTCA/DO-178B. Washington, DC: RTCA, December 1992.
- [41] Reason, J. *Human Error*. Cambridge University Press, Cambridge, UK, 1990.
- [42] Rushby, J. "Kernels for Safety?" *Safe and Secure Computing Systems*, T. Anderson Ed., Blackwell Scientific Publications, 1989.
- [43] 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.
- [44] Sha, L. "Using Simplicity to Control Complexity." *IEEE Software* 18(4):20-28, 2001.
- [45] Sha, L., R. Rajkumar and M. Gagliardi. "A Software Architecture for Dependable and Evolvable Industrial Computing Systems." Technical Report CMU/SEI-95-TR-005, Software Engineering Institute, Carnegie Mellon University, 1995.
- [46] 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.
- [47] Storey, N. *Safety-Critical Computer Systems*. Prentice Hall: Harlow, U.K., 1996.

- [48] Stewart, D.B., and G. Arora. "Dynamically Reconfigurable Embedded Software-Does It Make Sense?" Proc. Second IEEE International Conference on Engineering of Complex Computer Systems, Oct. 1996.
- [49] Strunk, E. A. *The Role of Natural Language in a Software Product*. M.S. Thesis, University of Virginia Dept. of Computer Science, May 2002.
- [50] Strunk, E. A., and J. C. Knight. "Assured Reconfiguration of Embedded Real-Time Software." Proc. International Conference on Dependable Systems and Networks (DSN 2004), Florence, Italy, June 2004.
- [51] Strunk, E. A., J. C. Knight, and M. A. Aiello. "Distributed Reconfigurable Avionics Architectures." Proc. 23rd Digital Avionics Systems Conference, Salt Lake City, UT, October 2004.
- [52] 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).
- [53] 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.
- [54] U.S. Department of Commerce, National Telecommunications and Information Administration, Institute for Telecommunications Services, Federal Std. 1037C.
- [55] 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.
- [56] Yeh, Y. C. "Triple-Triple Redundant 777 Primary Flight Computer." Proc. 1996 IEEE Aerospace Applications Conference, vol. 1, New York, N.Y., February 1996.

Appendix A

Architecture Specification



State and Operations

```

% In order to put requirements on data manipulation in an abstract manner,
% an abstract type system is created here. Ideally, a theory interpretation
% of this type system, including interpretations of uninterpreted types,
% will be built for each system.

% The data over which the system operates is modeled as a set of mappings from
% identifiers to values. This corresponds to global state for a system.

state : THEORY
BEGIN

% all system variables
data_id: NONEMPTY_TYPE

% all values that can be taken by any variable
data_value: TYPE = int

env_id: NONEMPTY_TYPE
env_param: NONEMPTY_TYPE

% implies no type rules, can restrict if needed
data_state: TYPE = [data_id -> data_value]

% the predicate and function types defined here allow restriction of access
% to global state, and essentially, support information hiding for the
% abstract model.

predicate(scope: set[data_id]) : TYPE =
{p: pred[data_state] | FORALL (st: data_state) :
  (p(st) =>
    FORALL (st2: data_state) :
      (FORALL (id: (scope)) : st2(id) = st(id)) => p(st2))}

func(scope: set[data_id]) : TYPE =
[# pre: predicate(scope),
  f: {f: [data_state -> data_state] |
    FORALL (d: data_state, id: data_id) :
      NOT scope(id) => f(d)(id) = d(id)}
#]

END state

```

Module Framework

```

% The module is the basic software component. Its predicates are defined,
% but no functions are defined specifically: the composition of module functions
% that is equivalent to application execution is modeled at the application level.

module : THEORY
BEGIN
IMPORTING state

% values for the module's service level parameter
module_svc: NONEMPTY_TYPE

% interpretation of implication over a module's restricted scope
implies(scope: set[data_id], sv: set[module_svc],
  p1, p2: [(sv) -> predicate((scope))] : bool =
  FORALL (service: (sv), st: data_state) : p1(service)(st) => p2(service)(st)

module_spec: TYPE =
[#
  % data elements the module can change
  scope: set[data_id],

  % service levels the module provides
  sv: set[module_svc],

  % data element that holds the current value of the module's service
  % level parameter
  svlvl_parm: (scope),

  % module invariant for each service level
  inv: [(sv) -> predicate((scope))],

  % module precondition for each service level
  pre: {p: [(sv) -> predicate((scope))] | implies(scope, sv, p, inv)},

  % module transition condition for each service level
  trans: {p: [(sv) -> predicate((scope))] | implies(scope, sv, p, inv)},

  % module postcondition for each service level
  post: {p: [(sv) -> predicate((scope))] | implies(scope, sv, p, inv)}
#]

END module

```

Application

```

application : THEORY
BEGIN
IMPORTING module

% configurations the applications can be in
app_svclvl : NONEMPTY_TYPE

% application scope: conjunction of module scopes
app_scope(modules: set[module_spec]) : set[data_id] =
  {d: data_id | EXISTS (m: (modules)) : m`scope(d)}

% mapping from module specs to module service levels
service_map(modules: set[module_spec]) : TYPE =
  {m: [(modules) -> module_svc] | FORALL (mod: (modules)) : mod`sv(m(mod))}

```

1. Predicates

```

% application postcondition: conjunction of module postconditions for the
% application's configuration
post(modules: set[module_spec], map: service_map(modules), st: data_state) :
  bool =
  FORALL (m: (modules)) : m`post(map(m))(st)

% application transition condition: conjunction of module transition conditions
% for the application's configuration
trans(modules: set[module_spec], map: service_map(modules), st: data_state) :
  bool =
  FORALL (m: (modules)) : m`trans(map(m))(st)

% application precondition: conjunction of module preconditions for the
% application's configuration
pre(modules: set[module_spec], map: service_map(modules), st: data_state) : bool =
  FORALL (m: (modules)) : m`pre(map(m))(st)

% application invariant: conjunction of module invariants for the
% application's configuration
inv(modules: set[module_spec], map: service_map(modules), st: data_state) : bool =
  FORALL (m: (modules)) : m`inv(map(m))(st)

```

2. Application type

```

% application specification
app_spec: TYPE =
[#
  % application configuration
  svcs: set[app_svclvl],

  % application's constituent modules
  modules: {f: finite_set[module_spec] |

```

```

nonempty?(f) AND
FORALL (m1, m2: (f)) : NOT (m1 = m2) =>
    NOT EXISTS (d: data_id) : m1`scope(d) AND m2`scope(d)},

% mapping from application configurations to module service levels
svcmmap: [(svcs) -> service_map(modules)],

% entry point for application functionality
execute: {f: [(svcs) -> func(app_scope(modules))] |
    FORALL (sv: (svcs), st: data_state) :
        inv(modules, svcmmap(sv), f(sv)`f(st))},

% application functionality in preparation for a possible reconfiguration
exec_halt: {f: [(svcs) -> func(app_scope(modules))] |
    FORALL (sv: (svcs), st: data_state) :
        inv(modules, svcmmap(sv), f(sv)`f(st))},

% preparation for reconfiguration if this application is
% the reconfiguration trigger
halt: {f: [(svcs) -> func(app_scope(modules))] |
    FORALL (sv: (svcs), st: data_state) :
        post(modules, svcmmap(sv), f(sv)`f(st))},

% function for the application to meet its transition condition
prep: {f: [(svcs) -> func(app_scope(modules))] |
    FORALL (sv: (svcs), st: data_state) :
        f(sv)`pre(st) = post(modules, svcmmap(sv), st) AND
        trans(modules, svcmmap(sv), f(sv)`f(st))}

#]

END application

```

SCRAM

```
% Defines the application-specific data that must be input to the SCRAM.
% Timing of the SCRAM is encoded in the required sequence of states
% in a system trace.
```

```
SCRAM: THEORY
BEGIN
IMPORTING application
```

1. Environment

```
% Basic type system for environmental factors and their possible values
valid_env: TYPE = [env_id ->set[env_param]]
env(v: valid_env) : TYPE =
  {f: [env_id -> env_param] | FORALL (e: env_id) : member(f(e), v(e))}
```

```
% Possible transitions from one reachable environmental state to another
env_txn(E: valid_env, D: set[env(E)]): TYPE =
[# source: (D),
  target: (D)
#]
```

```
% Collection of possible environmental states and transitions that affect which
% system transition is appropriate
reachable_env(E: valid_env): TYPE =
[# D: set[env(E)],
  txns: set[env_txn(E, D)]
#]
```

2. Transitions & SCRAM input

```
% Possible system configurations
speclvl: NONEMPTY_TYPE
```

```
% Model of real time
real_time: TYPE = nat
```

```
% Length of an execution cycle. System synchrony means that real time is only
% necessary in the environment model.
```

```
cycle: TYPE = nat
```

```
% Correspondence between execution cycles and the real time a cycle takes
cycle_time: real_time
```

```
% application - service level mapping that ensures the evaluation of the
% function is a member of the set of application configurations for the
% domain application
```

```
valid_app_svcs(apps: set[app_spec]) : TYPE =
  {f: [(apps) -> app_svclvl] | FORALL (app: (apps)) : app`svcs(f(app))}
```

```
% mapping from system configurations to application configurations
sys_configs(S: set[speclvl], apps: set[app_spec]) : TYPE =
```

```

    [(S) -> valid_app_svcs(apps)]

% system transitions
transition(apps: set[app_spec], S: set[speclvl], E: valid_env,
  R: reachable_env(E)) : TYPE =
[# source: (S),
  target: (S),
  % if need a precondition on state for trigger to activate, encode it as
  % an environment variable
  trigger: (R`D)
#]

% predicate ensuring a transition can be taken under any hypothesized condition
covering_txns(apps: set[app_spec], S: set[speclvl], E: valid_env, R:
  reachable_env(E),
  s: set[transition(apps, S, E, R)], primary: (S), start_env: set[(R`D)]): bool =
% Have to cover all transitions out of any start state
(FORALL (e: (start_env), t: (R`txns)) : t`source = e =>
  EXISTS (txn: (s)) : txn`source = primary AND txn`trigger = t`target) AND
% include start state here so that there can be self-transitions from it
(FORALL (source: (S), t_s: (s), d: (R`D)) :
  t_s`target = source AND t_s`trigger = d =>
  FORALL (t_e: (R`txns)) : t_e`source = d =>
    EXISTS (t_t: (s)) : t_t`trigger = t_e`target) AND
% transitions have to be deterministic
FORALL (source: (S), trigger: (R`D)) : NOT EXISTS (t1, t2: (s)) :
  t1 /= t2 AND t1`source = source AND t1`trigger = trigger AND
  t2`source = source AND t2`trigger = trigger

% collection of system configuration and transition data that must be input to
% the SCRAM
SCRAM_table(apps: set[app_spec], S: set[speclvl], E: valid_env, R:
  reachable_env(E)) :
  TYPE =
[# configs: sys_configs(S, apps),
  primary: (S),
  % set of configurations where an interrupt signal may not occur during
  % operation. Note that this set can be empty (e.g., in the case that interrupts
  % go between two configurations of equal value, depending on circumstances).
  safe: set[(S)],
  start_env: set[(R`D)],
  txns: {s: set[transition(apps, S, E, R)] |
    covering_txns(apps, S, E, R, s, primary, start_env)}
#]

% possible combinations of system and environmental states
sys_env_state(S: set[speclvl], E: valid_env, D: set[env(E)]) : TYPE =
[# s: (S),
  e: (D)
#]

% restricts the type above to states that could occur during operation
reachable_state(apps: set[app_spec], S: set[speclvl], E: valid_env,
  R: reachable_env(E), s_table: SCRAM_table(apps, S, E, R),
  s: (S), e: (R`D)) : bool =
  (s = s_table`primary AND s_table`start_env(e)) OR
  EXISTS (t: (s_table`txns)) : t`target = s AND t`trigger = e

END SCRAM

```

Reconfiguration Specification

```

reconf_spec: THEORY
BEGIN

IMPORTING SCRAM

% used to represent an inconsistent set of application configurations
indeterminate: speclvl

% ordering of applications, used to express execution dependencies
app_sequence(apps: finite_set[app_spec]) : TYPE =
  {s: finseq[(apps)] | s`length = card(apps) AND injective?(s`seq)}

```

1. Type

```

% reconfiguration specification
reconf_spec: TYPE =
[# % set of applications in the system
  apps: {appsp: finite_set[app_spec] |
    nonempty?(appsp) AND
    FORALL (app1, app2: (appsp)) : NOT (app1 = app2) =>
      % application scopes are disjoint
      NOT EXISTS (d: data_id) :
        app_scope(app1`modules) (d) AND app_scope(app2`modules) (d)},
  % system dependencies
  app_seq: app_sequence(apps),
  % set of configuration labels for the system
  S: set[{sp: speclvl | NOT sp = indeterminate}],
  % system environment type system
  E: valid_env,
  % possible environmental states
  R: reachable_env(E),
  % system-specific information on configurations and transitions
  SCRAM_info: {t: SCRAM_table(apps, S, E, R) | nonempty?(t`txns)},
  % function to choose a new configuration, if one is needed
  choose: {c: [(S), env((E)) -> (S)] |
    FORALL (s: (S), e: env((E))) :
      IF (NOT R`D(e)) THEN c(s, e) = s
      ELSIF NOT (reachable_state(apps, S, E, R, SCRAM_info, s, e)) THEN
        c(s, e) = s
      ELSE EXISTS (t: (SCRAM_info`txns)) :
        t`source = s AND t`trigger = e AND t`target = c(s, e)
      ENDIF},
  % time constraints on transitions
  T: [(S), (S) -> {t: real_time | t >= 4 * cycle_time}]
#]

```

2. Predicates

```

% composition functions for system predicates from application predicates
post(r: reconf_spec, svc: (r`S), st: data_state) : bool =

```

```
FORALL (app: (r`apps)) :
  post(app`modules, app`svcmmap(r`SCRAM_info`configs(svc)(app)), st)

trans(r: reconf_spec, svc: (r`S), st: data_state) : bool =
  FORALL (app: (r`apps)) :
    trans(app`modules, app`svcmmap(r`SCRAM_info`configs(svc)(app)), st)

pre(r: reconf_spec, svc: (r`S), st: data_state) : bool =
  FORALL (app: (r`apps)) :
    pre(app`modules, app`svcmmap(r`SCRAM_info`configs(svc)(app)), st)

inv(r: reconf_spec, svc: (r`S), st: data_state) : bool =
  FORALL (app: (r`apps)) :
    inv(app`modules, app`svcmmap(r`SCRAM_info`configs(svc)(app)), st)

END reconf_spec
```

Monitor

```
% defines application executions.
monitor: THEORY
BEGIN

IMPORTING reconf_spec
```

1. System State

```
% Reconfiguration status of an application
reconf_state: TYPE =
  {normal, interrupted, halting, exec_halting, prepping, training}

% Overall system state
sys_state : TYPE =
[# % reconfiguration specification for the system
  sp: reconf_spec,
  % system persistent storage
  st: data_state,
  % reconfiguration status for all applications
  reconf_st: [(sp`apps) -> reconf_state],
  % application configurations
  app_svclvls: valid_app_svcs(sp`apps),
  % last configuration each application was in
  app_last_svcs: valid_app_svcs(sp`apps),
  % system configuration
  svclvl: (sp`S),
  % system configuration during last SFTA
  last_svc: (sp`S),
  % time the state is true of the system
  t: real_time
#]

% lemma to help with TCC and other proofs
nonempty_apps: LEMMA
  FORALL (st: sys_state) :
    card(st`sp`apps) - 1 >= 0
```

2. Application Functions

```
% allows the system to be interrupted during execution.
success_exec?(st: sys_state) : {r: reconf_state | r = normal OR r = interrupted}
success_halt?(st: sys_state) : {r: reconf_state | r = halting OR r = interrupted}
success_eh?(st: sys_state) : {r: reconf_state | r = exec_halting OR
  r = interrupted}
success_prep?(st: sys_state) : {r: reconf_state | r = prepping OR r = interrupted}
success_train?(st: sys_state) : {r: reconf_state | r = training OR r = interrupted}

% check each function to see whether the current configuration is a safe one
% (i.e., no interrupt signals allowed)
execute(st: sys_state, app: (st`sp`apps)) : sys_state =
```

```

    LET new_st : data_state =
      app`execute(st`sp`SCRAM_info`configs(st`svclvl)(app))`f(st`st) IN
    IF (st`sp`SCRAM_info`safe(st`svclvl)) THEN st WITH [ `st := new_st]
    ELSIF st`reconf_st(app) = training THEN
      st WITH [ `st := new_st, `reconf_st(app) := success_train?(st)]
    ELSE
      st WITH [ `st := new_st, `reconf_st(app) := success_exec?(st)]
    ENDIF

halt(st: sys_state, app: (st`sp`apps)) : sys_state =
  LET new_st : data_state =
    app`halt(st`sp`SCRAM_info`configs(st`svclvl)(app))`f(st`st) IN
  IF (st`sp`SCRAM_info`safe(st`svclvl)) THEN st WITH [ `st := new_st]
  ELSE
    st WITH [ `st := new_st, `reconf_st(app) := success_halt?(st)]
  ENDIF

exec_halt(st: sys_state, app: (st`sp`apps)) : sys_state =
  LET new_st : data_state =
    app`exec_halt(st`sp`SCRAM_info`configs(st`svclvl)(app))`f(st`st) IN
  IF (st`sp`SCRAM_info`safe(st`svclvl)) THEN st WITH [ `st := new_st]
  ELSE st WITH [ `st := new_st, `reconf_st(app) := success_eh?(st)]
  ENDIF

prep(st: sys_state, app: (st`sp`apps)) : sys_state =
  LET new_st : data_state =
    app`prep(st`sp`SCRAM_info`configs(st`svclvl)(app))`f(st`st) IN
  IF (st`sp`SCRAM_info`safe(st`svclvl)) THEN st WITH [ `st := new_st]
  ELSE st WITH [ `st := new_st, `reconf_st(app) := success_prep?(st)]
  ENDIF

% application monitoring layer function
monitor(st: sys_state, app: (st`sp`apps)) : sys_state =
  CASES st`reconf_st(app) OF
    normal: execute(st, app),
    interrupted: st,
    halting: halt(st, app),
    exec_halting: exec_halt(st, app),
    prepping: prep(st, app),
    training: execute(st, app)
  ENDCASES

% lemmas to assist TCC proofs
monitor_apps_constant : LEMMA
  FORALL (st: sys_state, app: (st`sp`apps)) :
    monitor(st, app)`sp`apps = st`sp`apps

monitor_svcs_constant : LEMMA
  FORALL (st: sys_state, app: (st`sp`apps)) :
    monitor(st, app)`sp`S = st`sp`S

```

3. System Functions

```

% added to discharge recursive_monitor's TCC that is too strong and thus unprovable
% this could be shown of recursive_monitor if the function were not universally
% quantified (so monitor_apps_constant cannot be used in the proof)

```

```
rm_apps_const: AXIOM
  FORALL (st: sys_state, apps: finseq[(st`sp`apps)], n: below[apps`length],
    v: [{z: [st: sys_state, apps: finseq[(st`sp`apps)], below[apps`length]]
      | z`3 < n} -> sys_state]):
    NOT (n = 0) IMPLIES
      v(st, apps, n - 1)`sp`apps = st`sp`apps

% sequence of application executions that satisfy dependency requirements
recursive_monitor(st: sys_state, apps: finseq[(st`sp`apps)],
  n: below[apps`length]) :
RECURSIVE sys_state =
  IF (n = 0) THEN monitor(st, apps(0))
  ELSE monitor(recursive_monitor(st, apps, n-1), apps(n))
  ENDIF
MEASURE n

END monitor
```

Trace

```
% defines sequencing functions over system executions.
trace: THEORY
BEGIN

IMPORTING reconf_spec
```

1. System Functions

```
% lemmas used in TCC proofs
recursive_monitor_apps: LEMMA
  FORALL (st: sys_state, n: nat) :
    n >= card(st`sp`apps) OR
    recursive_monitor(st, st`sp`app_seq, n)`sp`apps = st`sp`apps

recursive_monitor_svclvl: LEMMA
  FORALL (st: sys_state, n: nat) :
    n >= card(st`sp`apps) OR
    recursive_monitor(st, st`sp`app_seq, n)`sp`S = st`sp`S

% function to define subsequent reconfiguration status of an application
% called when active configuration will remain the same
next_config(st: sys_state) : sys_state =
  st WITH [ `reconf_st :=
    % if there's a signal, set everything to halt;
    % take mismatched stages as a signal
    IF (EXISTS (app: (st`sp`apps)) : st`reconf_st(app) = interrupted OR
      (EXISTS (app1, app2: (st`sp`apps)) :
        st`reconf_st(app1) /= normal AND
        st`reconf_st(app1) /= normal AND
        st`reconf_st(app1) /= st`reconf_st(app1))) THEN
      LAMBDA (app: (st`sp`apps)) :
        IF st`reconf_st(app) /= normal THEN halting
        ELSE exec_halting
      ENDIF
    ELSE
      % leave as is if halting since system_monitor will take care of it
      IF (EXISTS (app: (st`sp`apps)) : st`reconf_st(app) = halting) THEN
        LAMBDA (app: (st`sp`apps)) : st`reconf_st(app)
      % change from prepping to training
      ELSIF (EXISTS (app: (st`sp`apps)) : st`reconf_st(app) = prepping) THEN
        LAMBDA (app: (st`sp`apps)) :
          IF (st`reconf_st(app) = prepping) THEN training
          ELSE normal
        ENDIF
      % change from training to normal, or everything is normal
      ELSE LAMBDA (app: (st`sp`apps)) : normal
      ENDIF
    ENDIF]

% coordination of application execution and reconfiguration status
system_monitor(st: sys_state, e: env(st`sp`E)) : sys_state =
  IF ((EXISTS (app: (st`sp`apps)) : st`reconf_st(app) = halting) AND
```

```

(NOT EXISTS (app: (st`sp`apps)) : st`reconf_st(app) = interrupted))
THEN LET (next_svc: (st`sp`S)) = st`sp`choose(st`svclvl, e) IN
  recursive_monitor(st WITH [
    `reconf_st :=
      (LAMBDA (app: (st`sp`apps)) :
        IF st`sp`SCRAM_info`configs(next_svc)(app) /=
          st`sp`SCRAM_info`configs(st`svclvl)(app)
        THEN prepping
        ELSE normal ENDIF),
    `app_svclvls :=
      (LAMBDA (app: (st`sp`apps)) :
        st`sp`SCRAM_info`configs(next_svc)(app)),
    `app_last_svcs :=
      (LAMBDA (app: (st`sp`apps)) : st`app_svclvls(app)),
    `last_svc := st`svclvl,
    `svclvl := next_svc], st`sp`app_seq, card(st`sp`apps)-1)
ELSE recursive_monitor(next_config(st), st`sp`app_seq, card(st`sp`apps)-1)
ENDIF

```

2. State Trace

```

% possible environmental sequences
valid_env_trace(E: valid_env, R: reachable_env(E)) : TYPE =
{e: [real_time -> env(E)] |
  FORALL (t1, t2: real_time) : t1 + 1 = t2 AND e(t1) /= e(t2) =>
    R`txns((# source := e(t1), target := e(t2) #))}

% 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 function application of system_monitor
%     to the previous state; and
% (4) the system state is synchronized with the environment.
sys_trace : TYPE =
[# sp: reconf_spec,
  env: valid_env_trace(sp`E, sp`R),
  tr: {c: [cycle -> {s: sys_state | s`sp = sp}] |
    (FORALL (c1, c2 : cycle) :
      c1 + 1 = c2 => c(c2) =
        system_monitor(c(c1), env(c1 * cycle_time))) AND
    (FORALL (app: (sp`apps)) : c(0)`reconf_st(app) = normal) AND
    inv(sp, c(0)`svclvl, c(0)`st) AND
    (FORALL (cyc1: cycle) : EXISTS (cyc2: cycle) : cyc2 > cyc1 AND
      FORALL (app: (sp`apps)) : c(cyc2)`reconf_st(app) = normal)}
#]

% encoding of the assumption that initialization takes only one execution cycle
train_time: AXIOM
  FORALL (s: sys_trace, c: cycle) :
    FORALL (app: (s`sp`apps)) :
      (s`tr(c)`reconf_st(app) = training =>
        pre(app`modules,
          app`svcmap(s`sp`SCRAM_info`configs(s`tr(c)`svclvl)(app)),
          s`tr(c)`st))

END trace

```

Lemmas

```

lemmas: THEORY
BEGIN

IMPORTING trace

reconfiguration: TYPE = [# start_c: cycle, end_c: cycle #]

% a reconfiguration is either bounded on one side by an interrupt
% and on the other by "normal" or bounded on both sides by an interrupt

reconfig_start?(s: sys_trace, c: cycle) : bool =
  EXISTS (app: (s`sp`apps)) : s`tr(c)`reconf_st(app) = interrupted

reconfig_end?(s: sys_trace, c: cycle) : bool =
  (FORALL (app: (s`sp`apps)) :
    (s`tr(c)`reconf_st(app) = normal OR s`tr(c)`reconf_st(app) = training)) OR
  (EXISTS (app: (s`sp`apps)) : s`tr(c)`reconf_st(app) = interrupted)

% whether the time is inside some reconfiguration round
reconfiguring?(s: sys_trace, c: cycle) : bool =
  EXISTS (app: (s`sp`apps)) : s`tr(c)`reconf_st(app) /= normal

get_reconfigs(s: sys_trace) : set[reconfiguration] =
  {r: reconfiguration |
    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))}

change_to_interrupt: LEMMA
  FORALL (st: sys_state, app: (st`sp`apps)) :
    FORALL (app2: (st`sp`apps)) :
      monitor(st, app)`reconf_st(app2) = st`reconf_st(app2) OR
      (app2 = app AND monitor(st, app)`reconf_st(app2) = interrupted)

change_to_interrupt_rec: LEMMA
  FORALL (st: sys_state, app: (st`sp`apps), n: nat) :
    (n >= card(st`sp`apps) OR
     recursive_monitor(st, st`sp`app_seq, n)`reconf_st(app) =
      st`reconf_st(app) OR
     recursive_monitor(st, st`sp`app_seq, n)`reconf_st(app) = interrupted)

m_speclvl_const : LEMMA
  FORALL (st: sys_state, app: (st`sp`apps)) : monitor(st, app)`svclvl = st`svclvl

rm_speclvl_const: LEMMA
  FORALL (st: sys_state, n: nat) :
    n >= card(st`sp`apps) OR
    recursive_monitor(st, st`sp`app_seq, n)`svclvl = st`svclvl

interrupt_st: LEMMA
  FORALL (s: sys_trace, c: cycle) :
    (EXISTS (app: (s`sp`apps)) : s`tr(c)`reconf_st(app) = interrupted) =>
    (FORALL (app: (s`sp`apps)) :
      s`tr(c+1)`reconf_st(app) = halting OR

```

```

s`tr(c+1)`reconf_st(app) = exec_halting OR
s`tr(c+1)`reconf_st(app) = interrupted)

reconf_halt: LEMMA
  FORALL (s: sys_trace, r: (get_reconfs(s))) :
    (FORALL (app: (s`sp`apps)) :
      (s`tr(r`start_c+1)`reconf_st(app) = halting OR
       s`tr(r`start_c+1)`reconf_st(app) = exec_halting OR
       s`tr(r`start_c+1)`reconf_st(app) = interrupted))

int_halt_st: LEMMA
  FORALL (s: sys_trace, r: (get_reconfs(s))) :
    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`start_c+1)`svclvl)(app)

int_halt_len: LEMMA
  FORALL (s: sys_trace, r: (get_reconfs(s))):
    (EXISTS (app: (s`sp`apps)) :
      s`tr(r`start_c+1)`reconf_st(app) = interrupted) =>
      r`end_c - r`start_c = 1

halt_st: LEMMA
  FORALL (s: sys_trace, c: cycle) :
    (EXISTS (app: (s`sp`apps)) : s`tr(c)`reconf_st(app) = halting) =>
      ((FORALL (app: (s`sp`apps)) :
        s`tr(c+1)`reconf_st(app) = prepping OR
        s`tr(c+1)`reconf_st(app) = normal OR
        s`tr(c+1)`reconf_st(app) = interrupted) OR
      EXISTS (app: (s`sp`apps)) : s`tr(c)`reconf_st(app) = interrupted)

reconf_prep: LEMMA
  FORALL (s: sys_trace, r: (get_reconfs(s))) :
    r`end_c - r`start_c > 1 =>
      (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`start_c+2)`svclvl)(app) AND
          s`tr(r`start_c+2)`reconf_st(app) = prepping) OR
         (s`sp`SCRAM_info`configs(s`tr(r`start_c)`svclvl)(app) =
          s`sp`SCRAM_info`configs(s`tr(r`start_c+2)`svclvl)(app) AND
          s`tr(r`start_c+2)`reconf_st(app) = normal) OR
          s`tr(r`start_c+2)`reconf_st(app) = interrupted))

prep_st: LEMMA
  FORALL (s: sys_trace, r: (get_reconfs(s))) :
    r`end_c - r`start_c > 2 =>
      (FORALL (app: (s`sp`apps)) :
        (s`tr(r`start_c+2)`reconf_st(app) = prepping =>
         (s`tr(r`start_c+3)`reconf_st(app) = training OR
          s`tr(r`start_c+3)`reconf_st(app) = interrupted)))

int_prep_len: LEMMA
  FORALL (s: sys_trace, r: (get_reconfs(s))):
    r`end_c - r`start_c > 1 =>
      ((EXISTS (app: (s`sp`apps)) :
        s`tr(r`start_c+2)`reconf_st(app) = interrupted) =>
        r`end_c - r`start_c = 2)

reconf_train: LEMMA

```

```

FORALL (s: sys_trace, r: (get_reconfigs(s))) :
  r`end_c - r`start_c > 2 =>
    (FORALL (app: (s`sp`apps)) :
      s`sp`SCRAM_info`configs(s`tr(r`start_c+3)`svclvl)(app) =
        s`sp`SCRAM_info`configs(s`tr(r`start_c+2)`svclvl)(app))

train_st: LEMMA
  FORALL (s: sys_trace, r: (get_reconfigs(s))) :
    r`end_c - r`start_c > 2 =>
      (FORALL (app: (s`sp`apps)) :
        (s`tr(r`start_c+3)`reconf_st(app) = training OR
          s`tr(r`start_c+3)`reconf_st(app) = normal OR
          s`tr(r`start_c+3)`reconf_st(app) = interrupted))

reconf_length: LEMMA
  FORALL (s: sys_trace, r: (get_reconfigs(s))) : r`end_c - r`start_c <= 3

invariant_monitor: LEMMA
  FORALL (st: sys_state, app: (st`sp`apps)) :
    (inv(st`sp, st`svclvl, st`st) =>
      inv(st`sp, st`svclvl, monitor(st, app)`st)) AND
    (NOT st`reconf_st(app) = interrupted =>
      inv(app`modules,
        app`svcmmap(st`sp`SCRAM_info`configs(st`svclvl)(app)),
        monitor(st, app)`st))

invariant_monitor_rec: LEMMA
  FORALL (st: sys_state, n: nat) :
    n >= card(st`sp`apps) OR
    (inv(st`sp, st`svclvl, st`st) =>
      inv(st`sp, st`svclvl,
        recursive_monitor(st, st`sp`app_seq, n)`st))

cycle_time: LEMMA
  FORALL (c: cycle) : c > 0 => c*cycle_time - cycle_time >= 0

same_conf_or_pre: LEMMA
  FORALL (s: sys_trace, r: (get_reconfigs(s)), 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) OR
    pre(app`modules, app`svcmmap(s`sp`SCRAM_info`configs
      (s`tr(r`end_c)`svclvl)(app)), s`tr(r`end_c)`st) OR
    (r`end_c - r`start_c < 3 AND
      EXISTS (app: (s`sp`apps)) :
        s`tr(r`end_c)`reconf_st(app) = interrupted) OR
    s`tr(r`end_c)`reconf_st(app) = interrupted
END lemmas

```

Invariant Lemmas

```

lemmas_inv: THEORY
BEGIN

IMPORTING lemmas

rm_spec_constant: LEMMA
  FORALL (st: sys_state, n: nat) :
    n >= card(st`sp`apps) OR
    recursive_monitor(st, st`sp`app_seq, n)`sp = st`sp

change_to_interrupt_rec_app: LEMMA
  FORALL (st: sys_state, app: (st`sp`apps), n: nat) :
    n >= card(st`sp`apps) OR
    ((NOT EXISTS (m: nat) : m <= n AND st`sp`app_seq(m) = app) =>
     recursive_monitor(st, st`sp`app_seq, n)`reconf_st(app) =
     st`reconf_st(app))

monitor_not_equal_nc: LEMMA
  FORALL (st: sys_state, app: (st`sp`apps), id: data_id) :
    (NOT app_scope(app`modules)(id) =>
     monitor(st, app)`st(id) = st`st(id))

monitor_inv_equal: LEMMA
  FORALL (st: sys_state, app: (st`sp`apps)) :
    st`reconf_st(app) /= interrupted =>
      (inv(app`modules,
         app`svcmmap(st`sp`SCRAM_info`configs(st`svclvl)(app)),
         monitor(st, app)`st))

inv_not_equal: LEMMA
  FORALL (st: sys_state, app: (st`sp`apps), n: posnat) :
    n >= card(st`sp`apps) OR
    (st`sp`app_seq(n) /= app =>
     (inv(app`modules,
        app`svcmmap(st`sp`SCRAM_info`configs(st`svclvl)(app)),
        recursive_monitor(st, st`sp`app_seq, n-1)`st) =>
      inv(app`modules,
         app`svcmmap(st`sp`SCRAM_info`configs(st`svclvl)(app)),
         recursive_monitor(st, st`sp`app_seq, n)`st)))

invariant_monitor_middle_app: LEMMA
  FORALL (st: sys_state, app: (st`sp`apps), n: nat) :
    n >= card(st`sp`apps) OR
    (((EXISTS (m: nat) : m <= n AND st`sp`app_seq(m) = app) AND
     st`reconf_st(app) /= interrupted) =>
     inv(app`modules,
        app`svcmmap(st`sp`SCRAM_info`configs(st`svclvl)(app)),
        recursive_monitor(st, st`sp`app_seq, n)`st))

invariant_monitor_rec_app: LEMMA
  FORALL (st: sys_state, app: (st`sp`apps)) :
    (NOT st`reconf_st(app) = interrupted) =>
      inv(app`modules,
         app`svcmmap(st`sp`SCRAM_info`configs(st`svclvl)(app)),

```

```
recursive_monitor(st, st`sp`app_seq, card(st`sp`apps)-1)`st)  
END lemmas_inv
```

Reconfiguration Properties

```

assured_reconfig: THEORY
BEGIN

IMPORTING lemmas_inv

% defines what makes up a reconfiguration. Essentially a repetition of the
% get_reconfigs function, repeated for clarity.
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))

% either: (1) a signal was generated before applications were notified of the new
% configuration and 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; at the end of the
% reconfiguration, the system will be in an appropriate new configuration
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))

% The reconfiguration takes less than or equal to its allotted 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)

% The function invariant holds
CP4: THEOREM
  FORALL (s: sys_trace, c: cycle) :
    % The function invariant holds
    inv(s`sp, s`tr(c)`svclvl, s`tr(c)`st) OR

    % A signal was generated in a function that was reconfiguring before
    % it met the transition condition for the new specification,
    % but while all applications were preparing to transition, so that
    % any application that generated a signal during that cycle will maintain
    % the invariant for the previous configuration, while any application that
    % did not generate a signal will maintain the invariant for the
    % new specification. Note that this case is always false if all applications
    % delay processing of signals during the cycle the transition condition
    % is being met.
    (c > 0 AND
      FORALL (app: (s`sp`apps)):

        % the application received 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`svcmmap(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`svcmmap(s`sp`SCRAM_info`configs(s`tr(c)`svclvl)(app)),
  s`tr(c)`st))

% set of predicates, one of which must be true at the end of a reconfiguration
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`svcmmap(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`svcmmap(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`svcmmap(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

END assured_reconfig

```

Appendix B

UAV System Specification



Example State

```

ex_state : THEORY
BEGIN

IMPORTING state

% This theory models the entire state of the system. It essentially emulates a
% type system that would normally be created as the base of a system specification,
% but which cannot be the base in this case because the type system is used to model
% the architecture rather than the computation.
% Thus, it is ugly and confusing, but it works.

```

1. Data values

```

cur_alt, cur_hdg, calc_left_aileron, calc_right_aileron, calc_rudder,
  calc_elevator, adjust_left_aileron, adjust_right_aileron, adjust_rudder,
  left_aileron, right_aileron, rudder, elevator, alt_hold, hdg_hold,
  tgt_alt, tgt_hdg, ap_fcs_cmd, elec_monitor, rudder_monitor : data_id

disabled, off, engaged, ah_only: data_value
battery, alternator: data_value
a_up, a_mid_up, a_neutral, a_mid_down, a_down: data_value
r_left, r_neutral, r_right, r_ho_left, r_ho_right: data_value
e_up, e_neutral, e_down: data_value
climb, descend, hold_alt, left, right, hold_hdg, nop: data_value

% status for autopilot altitude hold mode and heading hold mode
% off means operational but not engaged
ap_mode_status: set[data_value] =
  {v: data_value | v = disabled OR v = off OR v = engaged OR v = ah_only}

% status for alternator and battery in electrics subsystem
% backup means servicable but now working as backup
elec_status: set[data_value] =
  {v: data_value | v = battery OR v = alternator}

% status for ailerons, rudder, and elevator in flight control system
aileron_status: set[data_value] =
  {v: data_value | v = a_up OR v = a_mid_up OR v = a_neutral OR v = a_mid_down OR
  v = a_down}
rudder_status: set[data_value] =
  {v: data_value | v = r_left OR v = r_neutral OR v = r_right OR v = r_ho_left OR
  v = r_ho_right}
elevator_status: set[data_value] =
  {v: data_value | v = e_up OR v = e_neutral OR v = e_down}

% command type send from autopilot to FCS, nop for not specified command
cmd_type: set[data_value] =
  {v: data_value | v = climb OR v = descend OR v = hold_alt OR v = left OR
  v = right OR v = hold_hdg OR v = nop}

values_unique: AXIOM
battery /= alternator AND

```

```

a_up /= a_mid_up AND a_up /= a_neutral AND a_up /= a_mid_down AND a_up /= a_down
AND a_mid_up /= a_neutral AND a_mid_up /= a_mid_down AND a_mid_up /= a_down AND
a_neutral /= a_mid_down AND a_neutral /= a_down AND a_mid_down /= a_down AND

```

```

r_left /= r_neutral AND r_left /= r_right AND r_left /= r_ho_left AND r_left /=
r_ho_right AND r_neutral /= r_right AND r_neutral /= r_ho_left AND r_neutral /=
r_ho_right AND r_right /= r_ho_left AND r_right /= r_ho_right AND r_ho_left /=
r_ho_right AND

```

```

e_up /= e_neutral AND e_up /= e_down AND e_neutral /= e_down AND

```

```

climb /= descend AND climb /= hold_alt AND climb /= left AND climb /= right AND
climb /= hold_hdg AND climb /= nop AND descend /= hold_alt AND descend /= left AND
descend /= right AND descend /= hold_hdg AND descend /= nop AND hold_alt /= left
AND hold_alt /= right
AND hold_alt /= hold_hdg AND hold_alt /= nop AND left /= right AND left /= hold_hdg
AND left /= nop AND right /= hold_hdg AND right /= nop AND hold_hdg /= nop

```

2. Type encodings

```

type_constraints: AXIOM
  FORALL (st: data_state) :
    % for sensors, rely on int as the base type for state
    % int(st(cur_alt)) AND
    % int(st(cur_hdg)) AND

    % enumerated sensors values
    elec_status(st(elec_monitor)) AND
    rudder_status(st(rudder_monitor)) AND

    % for fcs
    aileron_status(st(calc_left_aileron)) AND
    aileron_status(st(calc_right_aileron)) AND
    rudder_status(st(calc_rudder)) AND
    elevator_status(st(calc_elevator)) AND

    aileron_status(st(adjust_left_aileron)) AND
    aileron_status(st(adjust_right_aileron)) AND
    rudder_status(st(adjust_rudder)) AND

    aileron_status(st(left_aileron)) AND
    aileron_status(st(right_aileron)) AND
    rudder_status(st(rudder)) AND
    elevator_status(st(elevator)) AND

    % command send from autopilot to FCS
    cmd_type(st(ap_fcs_cmd)) AND

    % for autopilot
    ap_mode_status(st(alt_hold)) AND
    ap_mode_status(st(hdg_hold)) AND
    %int?(tgt_alt) AND
    st(tgt_hdg) >= 0 AND st(tgt_hdg) < 360

END ex_state

```

Environment

```
environment: THEORY
BEGIN
```

```
IMPORTING SCRAM
```

```
% The nonequal axioms are again because we have simulated refinement rather than
% actually using it.
```

1. Environmental State Parameters

1.1. Electrical Subsystem State

```
electrics: env_id
alternator, battery: env_param
electrics_params : set[env_param] =
  {e: env_param | e = alternator OR e = battery}
```

1.2. Autopilot Subsystem State

```
autopilot: env_id
fullsvc, alt_hold_only, disabled: env_param
autopilot_params : set[env_param] =
  {e: env_param | e = fullsvc OR e = alt_hold_only OR e = disabled}
```

1.3. Rudder State

```
rudder: env_id
working, hard_over_left, hard_over_right: env_param
rudder_params : set[env_param] =
  {e: env_param | e = working OR e = hard_over_left OR e = hard_over_right}
```

```
different_env_params: AXIOM
alternator /= battery AND alternator /= fullsvc AND alternator /= alt_hold_only
AND alternator /= disabled AND alternator /= working AND alternator /=
hard_over_left AND alternator /= hard_over_right AND battery /= fullsvc AND
battery /= alt_hold_only AND battery /= disabled AND battery /= working AND battery
/= hard_over_left AND battery /= hard_over_right AND fullsvc /= alt_hold_only AND
fullsvc /= disabled AND fullsvc /= working AND fullsvc /= hard_over_left AND
fullsvc /= hard_over_right AND alt_hold_only /= disabled AND alt_hold_only /=
working AND alt_hold_only /= hard_over_left AND alt_hold_only /= hard_over_right
AND disabled /= working AND disabled /= hard_over_left AND disabled /=
hard_over_right AND working /= hard_over_left AND working /= hard_over_right AND
hard_over_left /= hard_over_right
```

2. Valid Environmental States

```
proto_env_id: set[env_id] =
  {e: env_id | e = autopilot OR e = electrics OR e = rudder}
```

```
proto_valid_env: valid_env = (LAMBDA (id: env_id) :
```

```

IF id = autopilot THEN
  {e: env_param | e = fullsvc OR e = alt_hold_only OR e = disabled}
ELSIF id = electrics THEN
  {e: env_param | e = alternator OR e = battery}
ELSE % rudder
  {e: env_param | e = working OR e = hard_over_left OR e = hard_over_right}
ENDIF)

proto_envs: set[env(proto_valid_env)] = {e: env(proto_valid_env) | true}

```

3. Possible Transitions

```

% only degrade - can't upgrade
degraded(source, target: env(proto_valid_env)) : bool =
  (source(electrics) = alternator AND target(electrics) = battery) OR
  (source(autopilot) = fullsvc AND target(autopilot) = alt_hold_only) OR
  (source(autopilot) = alt_hold_only AND target(autopilot) = disabled) OR
  (source(autopilot) = fullsvc AND target(autopilot) = disabled) OR
  (source(rudder) = working AND target(rudder) = hard_over_left) OR
  (source(rudder) = working AND target(rudder) = hard_over_right)

proto_env_txns: set[env_txn(proto_valid_env, proto_envs)] =
  {et: env_txn(proto_valid_env, proto_envs) | degraded(et`source, et`target)}

```

4. Reachable Environmental States

```

proto_reachable_env : reachable_env(proto_valid_env) =
  (# `D := proto_envs,
   `txns := proto_env_txns
  #)

END environment

```

Sensors

```
sensors : THEORY
BEGIN

IMPORTING application
IMPORTING ex_state
IMPORTING environment

% Sensors has one module:
% Sensors_update updates environmental input parameters (altitude, heading).
```

1. Sensors_update

1.1. Sensors_update Module Types

```
% Instantiations of the abstract system state types.
sensors_update_full : module_svc
sensors_update_svcs: set[module_svc] = {svc: module_svc | svc =
sensors_update_full}

sensors_update_svclvl : data_id
sensors_update_full : data_value

sensors_update_scope : set[data_id] =
    {d: data_id | d = cur_alt OR d = cur_hdg OR d = sensors_update_svclvl}
```

1.2. Sensors_update Module Functions

```
% Simple simulation of aircraft response to flight control surface positions.
% Both ailerons, elevator, and rudder centered: no change in altitude or heading
% Elevator up: altitude reduced 5 ft.
% Elevator down: altitude increased 5 ft.
% Ailerons are not currently used to increase aircraft altitude
% Left aileron up, right aileron down: heading change of 6 degrees counterclockwise
% Right aileron up, left aileron down: heading change of 6 degrees clockwise
% Left aileron half-up, right aileron half-down:
%   heading change of 3 degrees counterclockwise
% Right aileron half-up, left aileron half-down:
%   heading change of 3 degrees clockwise
% Rudder right: heading change of 3 degrees counterclockwise
% Rudder left: heading change of 3 degrees clockwise

% change is measured in degrees clockwise
heading_input_sim(st: data_state) : int =
    LET change: int =
        % contribution from ailerons
        IF (st(left_aileron) = a_up AND st(right_aileron) = a_down) THEN -6
        ELSIF (st(left_aileron) = a_down AND st(right_aileron) = a_up) THEN 6
        ELSIF (st(left_aileron) = a_mid_up AND st(right_aileron) = a_mid_down)
            THEN -3
        ELSIF (st(left_aileron) = a_mid_down AND st(right_aileron) = a_mid_up)
            THEN 3
        ELSE 0
    ENDIF
```

```

+
% contribution from rudder
IF st(rudder) = r_left THEN -3
ELSIF st(rudder) = r_right THEN 3
ELSE 0
ENDIF
IN st(cur_hdg) + change

% change is measured in altitude gain
altitude_input_sim(st: data_state) : int =
  LET change: int =
    % contribution from elevator
    IF st(elevator) = e_up THEN -5
    ELSIF st(elevator) = e_down THEN 5
    ELSE 0
    ENDIF
  IN st(cur_alt) + change

% sets environmental state to new calculated values
sensors_update_execute(sv: (sensors_update_svcs)) : func(sensors_update_scope) =
(# pre := (LAMBDA (st: data_state) : true),
  f := (LAMBDA (st: data_state) : st WITH
    [`cur_hdg := heading_input_sim(st),
     `cur_alt := altitude_input_sim(st)])
#)

```

1.3. Sensors_update Module

```

% defined to work accurately when called at any time
sensors_update: module_spec =
(# `scope := sensors_update_scope,
  `sv := sensors_update_svcs,
  `svclvl_parm := sensors_update_svclvl,
  `pre := (LAMBDA (sv: (sensors_update_svcs)) : (LAMBDA (s: data_state) : true)),
  `trans := (LAMBDA (sv: (sensors_update_svcs)) :
    (LAMBDA (s: data_state) : true)),
  `post := (LAMBDA (sv: (sensors_update_svcs)) :
    (LAMBDA (s: data_state) : true)),
  `inv := (LAMBDA (sv: (sensors_update_svcs)) : (LAMBDA (s: data_state) : true))
#)

```

2. Sensors Application

2.1. Application Structures

```

sensors_full : app_svclvl
sensors_svcs: set[app_svclvl] = {s: app_svclvl | s = sensors_full}
sensors_modules: finite_set[module_spec] = {m: module_spec | m = sensors_update}
sensors_scope: set[data_id] = app_scope(sensors_modules)

sensors_full_svcmap: service_map(sensors_modules) =
  (lambda (m: (sensors_modules)) : sensors_update_full)
sensors_svcmap: [(sensors_svcs) -> service_map(sensors_modules)] =
  (lambda (s: (sensors_svcs)) : sensors_full_svcmap)

```

2.2. Application Functions

```

sensors_execute(s: (sensors_svcs)) : func(sensors_scope) =
  (# pre := (LAMBDA (st: data_state) : true),
    f := (LAMBDA (st: data_state) :
      sensors_update_execute
        (sensors_svcmap(sensors_full)(sensors_update))`f(st))
  #)

sensors_exec_halt(s: (sensors_svcs)) : func(sensors_scope) =
  (# pre := (LAMBDA (st: data_state) : true),
    f := (LAMBDA (st: data_state) :
      sensors_update_execute
        (sensors_svcmap(sensors_full)(sensors_update))`f(st))
  #)

sensors_prep(s: (sensors_svcs)) : func(sensors_scope) =
  (# pre := (LAMBDA (st: data_state) : true),
    f := (LAMBDA (st: data_state) : st)
  #)

sensors_halt(s: (sensors_svcs)) : func(sensors_scope) =
  (# pre := (LAMBDA (st: data_state) : true),
    f := (LAMBDA (st: data_state) : st)
  #)

```

2.3. Application Type

```

% defined to work accurately at any time
sensors_app: app_spec =
  (# `svcs := sensors_svcs,
    `modules := sensors_modules,
    `svcmap := sensors_svcmap,
    `execute := sensors_execute,
    `exec_halt := sensors_exec_halt,
    `prep := sensors_prep,
    `halt := sensors_halt
  #)

END sensors

```

Pilot Interface

```

pilot_interface : THEORY
BEGIN

IMPORTING application
IMPORTING ex_state

% Pilot_interface has one module:
% Get_input reads any new commands from the pilot and updates state accordingly.

```

1. Get_input

1.1. Get_input Module Types

```

get_input_standard : module_svc
get_input_svcs: set[module_svc] = {svc: module_svc | svc = get_input_standard}
get_input_svclvl : data_id
get_input_standard : data_value

get_input_scope : set[data_id] =
  {d: data_id | d = alt_hold OR d = hdg_hold OR d = tgt_alt OR d = tgt_hdg OR
   d = get_input_svclvl}

```

1.2. Get_input Module Functions

```

% is altitude hold engaged?
altitude_hold(st: data_state) : bool
% is heading hold engaged?
heading_hold(st: data_state) : bool

% what is the current target altitude?
target_alt(st: data_state) : int
% what is the current target heading?
target_heading(st: data_state) : {i: int | i < 360}

get_input_execute(sv: (get_input_svcs)) : func(get_input_scope) =
(# pre := (LAMBDA (st: data_state) : true),
  f := (LAMBDA (st: data_state) : st WITH
    [ `alt_hold := altitude_hold(st),
      `hdg_hold := heading_hold(st),
      `tgt_alt := target_alt(st),
      `tgt_hdg := target_heading(st) ])
#)

```

1.3. Get_input Module

```

% can apply at any time
get_input: module_spec =
(# `scope := get_input_scope,
  `sv := get_input_svcs,
  `svclvl_parm := get_input_svclvl,
  `pre := (LAMBDA (sv: (get_input_svcs)) : (LAMBDA (s: data_state) : true)),
  `trans := (LAMBDA (sv: (get_input_svcs)) : (LAMBDA (s: data_state) : true)),
  `post := (LAMBDA (sv: (get_input_svcs)) : (LAMBDA (s: data_state) : true)),

```

```
`inv := (LAMBDA (sv: (get_input_svcs)) :(LAMBDA (s: data_state) : true))
#)
```

2. Pilot_interface Application

2.1. Application Structures

```
interface_standard : app_svclvl
interface_svcs: set[app_svclvl] = {s: app_svclvl | s = interface_standard}
interface_modules: finite_set[module_spec] = {m: module_spec | m = get_input}
interface_scope: set[data_id] = app_scope(interface_modules)

interface_standard_svcmap: service_map(interface_modules) =
  (LAMBDA (m: (interface_modules)) : get_input_standard)
interface_svcmap: [(interface_svcs) -> service_map(interface_modules)] =
  (LAMBDA (s: (interface_svcs)) : interface_standard_svcmap)
```

2.2. Application Functions

```
interface_execute(s: (interface_svcs)) : func(interface_scope) =
  (# pre := (LAMBDA (st: data_state) : true),
   f := (LAMBDA (st: data_state) :
         get_input_execute
         (interface_svcmap(interface_standard) (get_input)))`f(st))
#)

interface_exec_halt(s: (interface_svcs)) : func(interface_scope) =
  (# pre := (LAMBDA (st: data_state) : true),
   f := (LAMBDA (st: data_state) :
         get_input_execute
         (interface_svcmap(interface_standard) (get_input)))`f(st))
#)

interface_prep(s: (interface_svcs)) : func(interface_scope) =
  (# pre := (LAMBDA (st: data_state) : true),
   f := (LAMBDA (st: data_state) : st)
#)

interface_halt(s: (interface_svcs)) : func(interface_scope) =
  (# pre := (LAMBDA (st: data_state) : true),
   f := (LAMBDA (st: data_state) : st)
#)
```

2.3. Application Type

```
pilot_interface_app: app_spec =
  (# `svcs := interface_svcs,
   `modules := interface_modules,
   `svcmap := interface_svcmap,
   `execute := interface_execute,
   `exec_halt := interface_exec_halt,
   `prep := interface_prep,
   `halt := interface_halt
#)

END pilot_interface
```

Flight Control System Functional Specification

```
fcs_functionality : THEORY
BEGIN
```

```
IMPORTING ex_state
```

```
% FCS has three modules:
% FCS_calc: computes normal commands based on (human or autopilot) inputs
% FCS_adjust: adjusts those commands in a rudder hardover condition
% FCS_output: writes the calculated values to actuator state variables
```

1. FCS_calc

```
fcs_calc_svclvl : data_id

fcs_calc_scope : set[data_id] =
  {d: data_id | d = calc_left_aileron OR d = calc_right_aileron OR
    d = calc_rudder OR d = calc_elevator OR d = fcs_calc_svclvl}

% single service
fcs_calc_execute_standard : func(fcs_calc_scope) =
  (# pre := (LAMBDA (st: data_state) : true),
    f := (LAMBDA (st: data_state) :
      COND
        st(ap_fcs_cmd) = climb -> st WITH [calc_elevator := e_up],
        st(ap_fcs_cmd) = hold_alt -> st WITH [calc_elevator := e_neutral],
        st(ap_fcs_cmd) = descend -> st WITH [calc_elevator := e_down],
        st(ap_fcs_cmd) = left -> st WITH [calc_left_aileron := a_mid_up,
                                          calc_right_aileron := a_mid_down,
                                          calc_rudder := r_left],
        st(ap_fcs_cmd) = hold_hdg -> st WITH [calc_left_aileron := a_neutral,
                                              calc_right_aileron := a_neutral,
                                              calc_rudder := r_neutral],
        st(ap_fcs_cmd) = right -> st WITH [calc_left_aileron := a_mid_down,
                                           calc_right_aileron := a_mid_up,
                                           calc_rudder := r_right],
        ELSE -> st
      ENDCOND)
    #)
```

2. FCS_adjust

```
fcs_adjust_svclvl : data_id
fcs_adjust_scope : set[data_id] =
  {d: data_id | d = adjust_left_aileron OR d = adjust_right_aileron OR
    d = adjust_rudder OR d = fcs_adjust_svclvl}

% rudder is working properly, no adjustment needed
fcs_adjust_execute_nc: func(fcs_adjust_scope) =
  (# pre := (LAMBDA (st: data_state) : true),
    f := (LAMBDA (st: data_state) : st)
```

```

#)

% rudder hard over left
fcs_adjust_execute_rudder_ho_left : func(fcs_adjust_scope) =
(# pre := (LAMBDA (st: data_state) : true),
 f := (LAMBDA (st: data_state) :
  COND
    (st(calc_left_aileron) = a_mid_up AND
     st(calc_right_aileron) = a_mid_down) ->
      st WITH [adjust_left_aileron := a_neutral,
              adjust_right_aileron := a_neutral,
              adjust_rudder := left],
    (st(calc_left_aileron) = a_neutral AND
     st(calc_right_aileron) = a_neutral) ->
      st WITH [adjust_left_aileron := a_mid_down,
              adjust_right_aileron := a_mid_up,
              adjust_rudder := left],
    (st(calc_left_aileron) = a_mid_down AND
     st(calc_right_aileron) = a_mid_up) ->
      st WITH [adjust_left_aileron := a_down,
              adjust_right_aileron := a_up,
              adjust_rudder := left],
    ELSE -> st
  ENDCOND)
#)

% rudder hard over right
fcs_adjust_execute_rudder_ho_right : func(fcs_adjust_scope) =
(# pre := (LAMBDA (st: data_state) : true),
 f := (LAMBDA (st: data_state) :
  COND
    (st(calc_left_aileron) = a_mid_up AND
     st(calc_right_aileron) = a_mid_down) ->
      st WITH [adjust_left_aileron := a_up,
              adjust_right_aileron := a_down,
              adjust_rudder := right],
    (st(calc_left_aileron) = a_neutral AND
     st(calc_right_aileron) = a_neutral) ->
      st WITH [adjust_left_aileron := a_mid_up,
              adjust_right_aileron := a_mid_down,
              adjust_rudder := right],
    (st(calc_left_aileron) = a_mid_down AND
     st(calc_right_aileron) = a_mid_up) ->
      st WITH [adjust_left_aileron := a_neutral,
              adjust_right_aileron := a_neutral,
              adjust_rudder := right],
    ELSE -> st
  ENDCOND)
#)

```

3. FCS_output

```

% writes the appropriate output (normal or adjusted) to the actuators.
fcs_output_svclvl: data_id
fcs_output_scope : set[data_id] =
  {d: data_id | d = left_aileron OR d = right_aileron OR d = rudder OR

```

```
        d = elevator OR d = fcs_output_svclvl}

% not adjusted
fcs_output_standard: func(fcs_output_scope) =
(# pre := (LAMBDA (st: data_state) : true),
  f := (LAMBDA (st: data_state) : st WITH
        [elevator := st(calc_elevator),
         left_aileron := st(calc_left_aileron),
         right_aileron := st(calc_right_aileron),
         rudder := st(calc_rudder)]))
#)

% adjusted
fcs_output_rudder_ho: func(fcs_output_scope) =
(# pre := (LAMBDA (st: data_state) : true),
  f := (LAMBDA (st: data_state) : st WITH
        [elevator := st(calc_elevator),
         left_aileron := st(adjust_left_aileron),
         right_aileron := st(adjust_right_aileron),
         rudder := st(adjust_rudder)]))
#)

END fcs_functionality
```

Flight Control System

```
fcs : THEORY
BEGIN

IMPORTING application
IMPORTING fcs_functionality

% FCS has three modules:
% FCS_calc: computes normal commands based on (human or autopilot) inputs
% FCS_adjust: adjusts those commands in a rudder hardover condition.
% FCS_output: writes the calculated values to actuator state variables
```

1. FCS_calc

1.1. FCS_calc Module Types

```
% FCS_calc has only a single service, since FCS_adjust will alter values for
% alternative services
fcs_calc_standard: module_svc
fcs_calc_standard: data_value
fcs_calc_svcs: set[module_svc] = {svc: module_svc | svc = fcs_calc_standard}
```

1.2. FCS_calc Module Functions

```
% maps to the functionality specification
fcs_calc_execute(sv: (fcs_calc_svcs)) : func(fcs_calc_scope) =
fcs_calc_execute_standard
```

1.3. FCS_calc Module

```
% effectively no consistency conditions for the control surfaces
fcs_calc: module_spec =
(# `scope := fcs_calc_scope,
  `sv := fcs_calc_svcs,
  `svclvl_parm := fcs_calc_svclvl,
  `pre := (LAMBDA (sv: (fcs_calc_svcs)) : (LAMBDA (s: data_state) : true)),
  `trans := (LAMBDA (sv: (fcs_calc_svcs)) : (LAMBDA (s: data_state) : true)),
  `post := (LAMBDA (sv: (fcs_calc_svcs)) : (LAMBDA (s: data_state) : true)),
  `inv := (LAMBDA (sv: (fcs_calc_svcs)) : (LAMBDA (s: data_state) : true))
#)
```

2. FCS_adjust

2.1. FCS_adjust Module Types

```
fcs_adjust_nc, fcs_adjust_ho_left, fcs_adjust_ho_right: module_svc
fcs_adjust_svcs: set[module_svc] =
  {svc: module_svc | svc = fcs_adjust_nc OR svc = fcs_adjust_ho_left OR
  svc = fcs_adjust_ho_right}
```

```
fcs_adjust_svclvl: data_id
fcs_adjust_nc, fcs_adjust_ho_left, fcs_adjust_ho_right: data_value
```

2.2. FCS_adjust Module Functions

```
fcs_adjust_execute(sv: (fcs_adjust_svcs)) : func(fcs_adjust_scope) =
  IF sv = fcs_adjust_ho_left THEN
    fcs_adjust_execute_rudder_ho_left
  ELSIF sv = fcs_adjust_ho_right THEN
    fcs_adjust_execute_rudder_ho_right
  ELSE % sv = fcs_adjust_nc
    fcs_adjust_execute_nc
  ENDIF
```

2.3. FCS_adjust Module

```
% effectively no consistency conditions for the control surfaces
fcs_adjust: module_spec =
(# `scope := fcs_adjust_scope,
  `sv := fcs_adjust_svcs,
  `svclvl_parm := fcs_adjust_svclvl,
  `pre := (LAMBDA (sv: (fcs_adjust_svcs)) : (LAMBDA (s: data_state) : true)),
  `trans := (LAMBDA (sv: (fcs_adjust_svcs)) : (LAMBDA (s: data_state) : true)),
  `post := (LAMBDA (sv: (fcs_adjust_svcs)) : (LAMBDA (s: data_state) : true)),
  `inv := (LAMBDA (sv: (fcs_adjust_svcs)) : (LAMBDA (s: data_state) : true))
#)
```

3. FCS_output

3.1. FCS_output Module Types

```
% FCS_output has two services:
% FCS_output_standard: applies only FCS_calc's computations
% FCS_output_ho: adjusts the calculated outputs for the hard-over condition
fcs_output_standard, fcs_output_ho: module_svc
fcs_output_standard, fcs_output_ho: data_value
fcs_output_svcs: set[module_svc] =
  {svc: module_svc | svc = fcs_output_standard OR svc = fcs_output_ho}
```

3.2. FCS_output Module Functions

```
% maps to the appropriate functionality specification
fcs_output_execute(sv: (fcs_output_svcs)) : func(fcs_output_scope) =
  IF sv = fcs_output_ho THEN fcs_output_rudder_ho
  ELSE fcs_output_standard
  ENDIF
```

3.3. FCS_output Module

```
% effectively no consistency conditions for the control surfaces
fcs_output: module_spec =
(# `scope := fcs_output_scope,
  `sv := fcs_output_svcs,
  `svclvl_parm := fcs_output_svclvl,
  `pre := (LAMBDA (sv: (fcs_output_svcs)) : (LAMBDA (s: data_state) : true)),
  `trans := (LAMBDA (sv: (fcs_output_svcs)) : (LAMBDA (s: data_state) : true)),
```

```

`post := (LAMBDA (sv: (fcs_output_svcs)) : (LAMBDA (s: data_state) : true)),
`inv := (LAMBDA (sv: (fcs_output_svcs)) : (LAMBDA (s: data_state) : true))
#)

```

4. FCS Application

4.1. Application Structures

```

fcs_standard, fcs_rudder_ho_left, fcs_rudder_ho_right : app_svclvl
fcs_svcs: set[app_svclvl] =
  {s: app_svclvl | s = fcs_standard OR s = fcs_rudder_ho_left OR
   s = fcs_rudder_ho_right}
fcs_modules: finite_set[module_spec] =
  {m: module_spec | m = fcs_calc OR m = fcs_adjust OR m = fcs_output}
fcs_scope: set[data_id] = app_scope(fcs_modules)

fcs_standard_svcmap: service_map(fcs_modules) =
  (LAMBDA (m: (fcs_modules)) :
   IF (m = fcs_calc) THEN fcs_calc_standard
   ELSIF (m = fcs_adjust) THEN fcs_adjust_nc
   ELSE fcs_output_standard
   ENDIF)
fcs_ho_left_svcmap: service_map(fcs_modules) =
  (LAMBDA (m: (fcs_modules)) :
   IF (m = fcs_calc) THEN fcs_calc_standard
   ELSIF (m = fcs_adjust) THEN fcs_adjust_ho_left
   ELSE fcs_output_ho
   ENDIF)
fcs_ho_right_svcmap: service_map(fcs_modules) =
  (LAMBDA (m: (fcs_modules)) :
   IF (m = fcs_calc) THEN fcs_calc_standard
   ELSIF (m = fcs_adjust) THEN fcs_adjust_ho_right
   ELSE fcs_output_ho
   ENDIF)
fcs_svcmap: [(fcs_svcs) -> service_map(fcs_modules)] =
  (LAMBDA (sv: (fcs_svcs)) :
   IF (sv = fcs_rudder_ho_left) THEN fcs_ho_left_svcmap
   ELSIF (sv = fcs_rudder_ho_right) THEN fcs_ho_right_svcmap
   ELSE fcs_standard_svcmap
   ENDIF)

```

4.2. Application Functions

```

% normal function
fcs_execute(s: (fcs_svcs)) : func(fcs_scope) =
  (# pre := (LAMBDA (st: data_state) :
   fcs_calc_execute(fcs_svcmap(s) (fcs_calc))`pre(st) AND
   fcs_adjust_execute(fcs_svcmap(s) (fcs_adjust))`pre(st) AND
   fcs_output_execute(fcs_svcmap(s) (fcs_output))`pre(st)),
   f := (LAMBDA (st: data_state) :
   fcs_output_execute(fcs_svcmap(s) (fcs_output))`f
   (fcs_adjust_execute(fcs_svcmap(s) (fcs_adjust))`f
   (fcs_calc_execute(fcs_svcmap(s) (fcs_calc))`f(st))))
  #)

% same as execute

```

```

fcs_exec_halt(s: (fcs_svcs)) : func(fcs_scope) =
  (# pre := (LAMBDA (st: data_state) :
    fcs_calc_execute(fcs_svcmap(s) (fcs_calc))`pre(st) AND
    fcs_adjust_execute(fcs_svcmap(s) (fcs_adjust))`pre(st) AND
    fcs_output_execute(fcs_svcmap(s) (fcs_output))`pre(st)),
  f := (LAMBDA (st: data_state) :
    fcs_output_execute(fcs_svcmap(s) (fcs_output))`f
    (fcs_adjust_execute(fcs_svcmap(s) (fcs_adjust))`f
    (fcs_calc_execute(fcs_svcmap(s) (fcs_calc))`f(st))))
  #)

% change service level parameters to specify new function, otherwise no change
fcs_prep(s: (fcs_svcs)) : func(fcs_scope) =
  % rudder hard over right
  IF s = fcs_rudder_ho_right THEN
    (# pre := (LAMBDA (st: data_state) : true),
    f := (LAMBDA (st: data_state) : st WITH
      [ fcs_calc_svclvl := fcs_calc_standard,
        fcs_adjust_svclvl := fcs_adjust_ho_right,
        fcs_output_svclvl := fcs_output_ho])
    #)
  % rudder hard over left
  ELSIF s = fcs_rudder_ho_left THEN
    (# pre := (LAMBDA (st: data_state) : true),
    f := (LAMBDA (st: data_state) : st WITH
      [ fcs_calc_svclvl := fcs_calc_standard,
        fcs_adjust_svclvl := fcs_adjust_ho_left,
        fcs_output_svclvl := fcs_output_ho])
    #)
  % standard
  ELSE
    (# pre := (LAMBDA (st: data_state) : true),
    f := (LAMBDA (st: data_state) : st WITH
      [ fcs_calc_svclvl := fcs_calc_standard,
        fcs_adjust_svclvl := fcs_adjust_nc,
        fcs_output_svclvl := fcs_output_standard])
    #)
  ENDIF

% no execution and no other change
fcs_halt(s: (fcs_svcs)) : func(fcs_scope) =
  (# pre := (LAMBDA (st: data_state) : true),
  f := (LAMBDA (st: data_state) : st)
  #)

```

4.3. Application Type

```

fcs_app: app_spec =
  (# `svcs := fcs_svcs,
    `modules := fcs_modules,
    `svcmap := fcs_svcmap,
    `execute := fcs_execute,
    `exec_halt := fcs_exec_halt,
    `prep := fcs_prep,
    `halt := fcs_halt
  #)

END fcs

```

Autopilot Functionality

```

autopilot_functionality : THEORY
BEGIN

IMPORTING application
IMPORTING ex_state

% The autopilot has one module:
% AP_compute: computes command to send to the FCS
% based on user input and sensor data

```

1. AP_compute

```

ap_compute_svclvl: data_id
ap_compute_standard, ap_compute_ah_only, ap_compute_off: data_value

ap_compute_scope: set[data_id] =
  {d: data_id | d = ap_fcs_cmd OR d = ap_compute_svclvl}

% give altitude changes precedence over heading changes
% since the FCS cannot effect both changes simultaneously

% 5 ft. difference in altitude or 6 degree difference in heading required to make
% adjustments so that implementation will not flutter around a value
ap_compute_execute: func(ap_compute_scope) =
  (# pre := (LAMBDA (st: data_state) : true),
   f := (LAMBDA (st: data_state) :
     IF st(ap_compute_svclvl) = ap_compute_standard THEN
       % full service
       st WITH
         [ `ap_fcs_cmd :=
           IF (st(alt_hold) = engaged AND
              st(tgt_alt) > st(cur_alt) + 5) THEN climb
           ELSIF (st(alt_hold) = engaged AND
                 st(tgt_alt) < st(cur_alt) - 5) THEN descend
           ELSIF (st(hdg_hold) = engaged AND
                 ((st(tgt_hdg) < st(cur_hdg) - 6 AND
                  st(cur_hdg) - st(tgt_hdg) < 180)
                  OR
                 (st(tgt_hdg) > st(cur_hdg) + 6 AND
                  st(tgt_hdg) - st(cur_hdg) > 180))) THEN left
           ELSIF (st(hdg_hold) = engaged AND
                 ((st(tgt_hdg) > st(cur_hdg) + 6 AND
                  st(tgt_hdg) - st(cur_hdg) < 180)
                  OR
                 (st(tgt_hdg) < st(cur_hdg) - 6 AND
                  st(cur_hdg) - st(tgt_hdg) > 180))) THEN right
           ELSIF st(alt_hold) = engaged THEN hold_alt
           ELSIF st(hdg_hold) = engaged THEN hold_hdg
           ELSE nop
           ENDIF]
     ELSIF st(ap_compute_svclvl) = ap_compute_ah_only THEN

```

```

% altitude hold only
st WITH
  [`ap_fcs_cmd :=
    IF (st(alt_hold) = engaged AND
        st(tgt_alt) > st(cur_alt) + 5) THEN climb
    ELSIF (st(alt_hold) = engaged AND
        st(tgt_alt) < st(cur_alt) - 5) THEN descend
    ELSIF st(alt_hold) = engaged THEN hold_alt
    ELSE nop
    ENDIF]

ELSE
  % off
  st WITH [`ap_fcs_cmd := nop]

ENDIF
)
#)

% give altitude changes precedence over heading changes
% since the FCS cannot effect both changes simultaneously
ap_compute_exec_halt: func(ap_compute_scope) =
(# pre := (LAMBDA (st: data_state) : true),
 f := (LAMBDA (st: data_state) :
  IF st(ap_compute_svclvl) = ap_compute_standard THEN
    % full service
    st WITH
      [`ap_fcs_cmd :=
        IF (st(alt_hold) = engaged AND
            st(tgt_alt) > st(cur_alt) + 5) THEN climb
        ELSIF (st(alt_hold) = engaged AND
            st(tgt_alt) < st(cur_alt) - 5) THEN descend
        ELSIF (st(hdg_hold) = engaged AND
            ((st(tgt_hdg) < st(cur_hdg) - 6 AND
            st(cur_hdg) - st(tgt_hdg) < 180)
            OR
            (st(tgt_hdg) > st(cur_hdg) + 6 AND
            st(tgt_hdg) - st(cur_hdg) > 180))) THEN left
        ELSIF (st(hdg_hold) = engaged AND
            ((st(tgt_hdg) > st(cur_hdg) + 6 AND
            st(tgt_hdg) - st(cur_hdg) < 180)
            OR
            (st(tgt_hdg) < st(cur_hdg) - 6 AND
            st(cur_hdg) - st(tgt_hdg) > 180))) THEN right
        ELSIF st(alt_hold) = engaged THEN hold_alt
        ELSIF st(hdg_hold) = engaged THEN hold_hdg
        ELSE nop
        ENDIF]

    ELSIF st(ap_compute_svclvl) = ap_compute_ah_only THEN
      % altitude hold only
      st WITH
        [`ap_fcs_cmd :=
          IF (st(alt_hold) = engaged AND
              st(tgt_alt) > st(cur_alt) + 5) THEN climb
          ELSIF (st(alt_hold) = engaged AND
              st(tgt_alt) < st(cur_alt) - 5) THEN descend
          ELSIF st(alt_hold) = engaged THEN hold_alt
          ELSE nop
          ENDIF]

```

```
                ENDIF]

ELSE
    % off
    st WITH [`ap_fcs_cmd := nop]

ENDIF)

#)

END autopilot_functionality
```

Autopilot

```

autopilot : THEORY
BEGIN

IMPORTING autopilot_functionality
% The autopilot has one module:
% AP_compute: computes command to send to the FCS
% based on user input and sensor data

```

1. AP_compute Module

```

ap_compute_standard, ap_compute_ah_only, ap_compute_off: module_svc
ap_compute_svcs: set[module_svc] =
    {m: module_svc | m = ap_compute_standard OR m = ap_compute_ah_only OR
      m = ap_compute_off}

% the autopilot is turned off when a transition occurs
% any function that the autopilot can no longer maintain is disabled
ap_compute_module: module_spec =
(# scope := ap_compute_scope,
  sv := ap_compute_svcs,
  svclvl_parm := ap_compute_svclvl,

  % precondition must hold after first operation cycle: at this point, the
  % autopilot may be sending values
  pre := (LAMBDA (sv: (ap_compute_svcs)) :
    IF sv = ap_compute_standard THEN (LAMBDA (s: data_state) : true)
    ELSIF sv = ap_compute_ah_only THEN
      (LAMBDA (s: data_state) :
        s(ap_fcs_cmd) = climb OR s(ap_fcs_cmd) = descend OR
        s(ap_fcs_cmd) = nop)
      % ap_none_svc
    ELSE (LAMBDA (s: data_state) : s(ap_fcs_cmd) = nop)
    ENDIF),

  % autopilot does not send commands to fcs during transition
  trans := (LAMBDA (sv: (ap_compute_svcs)) :
    (LAMBDA (s: data_state) : s(ap_fcs_cmd) = nop)),

  post := (LAMBDA (sv: (ap_compute_svcs)) :
    (LAMBDA (s: data_state) : s(ap_fcs_cmd) = nop)),

  % disallows unachievable commands
  inv := (LAMBDA (sv: (ap_compute_svcs)) :
    IF sv = ap_compute_standard THEN (LAMBDA (s: data_state) : true)
    ELSIF sv = ap_compute_ah_only THEN
      (LAMBDA (s: data_state) :
        s(ap_fcs_cmd) = climb OR s(ap_fcs_cmd) = descend OR
        s(ap_fcs_cmd) = nop)
      % ap_none_svc
    ELSE (LAMBDA (s: data_state) : s(ap_fcs_cmd) = nop)
    ENDIF)
#)

```

2. Autopilot Application

2.1. Application Structures

```

ap_standard, ap_ah_only, ap_off: app_svclvl
ap_svcs: set[app_svclvl] =
    {s: app_svclvl | s = ap_standard OR s = ap_ah_only OR s = ap_off}
ap_modules: finite_set[module_spec] = {m: module_spec | m = ap_compute_module}
ap_scope: set[data_id] = app_scope(ap_modules)

ap_standard_svcmap: service_map(ap_modules) =
    (lambda (m: (ap_modules)) : ap_compute_standard)
ap_ah_only_svcmap: service_map(ap_modules) =
    (lambda (m: (ap_modules)) : ap_compute_ah_only)
ap_off_svcmap: service_map(ap_modules) =
    (lambda (m: (ap_modules)) : ap_compute_off)
ap_svcmap: [(ap_svcs) -> service_map(ap_modules)] =
    (lambda (s: (ap_svcs)) :
        IF s = ap_standard THEN ap_standard_svcmap
        ELSIF s = ap_ah_only THEN ap_ah_only_svcmap
        ELSE ap_off_svcmap
        ENDIF)

```

2.2. Application functions

```
% sends no commands to FCS
```

2.2.1. Prep

```

ap_prep_full: func(ap_scope) =
    (# pre := (LAMBDA (st: data_state) : true),
     f := (LAMBDA (st: data_state) : st WITH
           [ `ap_fcs_cmd := nop,
             `ap_compute_svclvl := ap_compute_standard])
    #)

```

```

ap_prep_ah_only: func(ap_scope) =
    (# pre := (LAMBDA (st: data_state) : true),
     f := (LAMBDA (st: data_state) : st WITH
           [ `ap_fcs_cmd := nop,
             `ap_compute_svclvl := ap_compute_ah_only])
    #)

```

```

ap_prep_none: func(ap_scope) =
    (# pre := (LAMBDA (st: data_state) : true),
     f := (LAMBDA (st: data_state) : st WITH
           [ `ap_fcs_cmd := nop,
             `ap_compute_svclvl := ap_compute_off])
    #)

```

2.2.2. Halt

```

ap_halt: func(ap_scope) =
    (# pre := (LAMBDA (st: data_state) : true),
     f := (LAMBDA (st: data_state) : st WITH [`ap_fcs_cmd := nop])
    #)

```

2.3. Application Type

```
autopilot_app: app_spec =
(# `svcs := ap_svcs,
  `modules := ap_modules,
  `svcmmap := ap_svcmmap,
  `execute := (lambda (sv: (ap_svcs)) : ap_compute_execute),
  `exec_halt := (lambda (sv: (ap_svcs)) : ap_compute_exec_halt),
  `prep := (lambda (sv: (ap_svcs)) :
    IF sv = ap_standard THEN ap_prep_full
    ELSIF sv = ap_ah_only THEN ap_prep_ah_only
    % ap_none_svc
    ELSE ap_prep_none
    ENDIF),
  `halt := (lambda (sv: (ap_svcs)) : ap_halt)
#)

END autopilot
```

Prototype System Reconfiguration Specification

```

prototype_reconf_spec: THEORY
BEGIN

IMPORTING reconf_spec
IMPORTING sensors, pilot_interface, fcs, autopilot
IMPORTING environment

% applications in the prototype system
proto_apps: set[app_spec] =
  {app: app_spec | app = sensors_app OR app = pilot_interface_app OR
    app = autopilot_app OR app = fcs_app}

% to make things easier since PVS compares equality by structure rather than name
different_apps: AXIOM
sensors_app /= pilot_interface_app AND sensors_app /= autopilot_app AND
sensors_app /= fcs_app AND pilot_interface_app /= autopilot_app AND
pilot_interface_app /= fcs_app AND autopilot_app /= fcs_app

```

1. System Configurations

```

% primary specification
full_service: speclvl

% partial autopilot malfunction, sufficient power, rudder OK
alt_hold_only: speclvl

% complete autopilot malfunction or battery power, rudder OK
fcs_only: speclvl

% working autopilot, sufficient power, rudder hard-over left
rudder_ho_left : speclvl

% working autopilot, sufficient power, rudder hard-over right
rudder_ho_right : speclvl

% partially malfunctioning autopilot, sufficient power, rudder hard-over left
rudder_ho_left_ah_only : speclvl

% partially malfunctioning autopilot, sufficient power, rudder hard-over right
rudder_ho_right_ah_only : speclvl

% broken autopilot, sufficient power, rudder hard-over left
rudder_ho_left_fcs_only : speclvl

% broken autopilot, sufficient power, rudder hard-over right
rudder_ho_right_fcs_only : speclvl

% yet another hack
different_lvls: AXIOM
  full_service /= alt_hold_only AND full_service /= fcs_only AND
    full_service /= rudder_ho_left AND full_service /= rudder_ho_right AND
    full_service /= rudder_ho_left_ah_only AND

```

```

    full_service /= rudder_ho_right_ah_only AND
    full_service /= rudder_ho_left_fcs_only AND
    full_service /= rudder_ho_right_fcs_only AND
    full_service /= indeterminate AND
alt_hold_only /= fcs_only AND alt_hold_only /= rudder_ho_left AND
    alt_hold_only /= rudder_ho_right AND
    alt_hold_only /= rudder_ho_left_ah_only AND
    alt_hold_only /= rudder_ho_right_ah_only AND
    alt_hold_only /= rudder_ho_left_fcs_only AND
    alt_hold_only /= rudder_ho_right_fcs_only AND
    alt_hold_only /= indeterminate AND
fcs_only /= rudder_ho_left AND fcs_only /= rudder_ho_right AND
    fcs_only /= rudder_ho_left_ah_only AND
    fcs_only /= rudder_ho_right_ah_only AND
    fcs_only /= rudder_ho_left_fcs_only AND
    fcs_only /= rudder_ho_right_fcs_only AND
    fcs_only /= indeterminate AND
rudder_ho_left /= rudder_ho_right AND
    rudder_ho_left /= rudder_ho_left_ah_only AND
    rudder_ho_left /= rudder_ho_right_ah_only AND
    rudder_ho_left /= rudder_ho_left_fcs_only AND
    rudder_ho_left /= rudder_ho_right_fcs_only AND
    rudder_ho_left /= indeterminate AND
rudder_ho_right /= rudder_ho_left_ah_only AND
    rudder_ho_right /= rudder_ho_right_ah_only AND
    rudder_ho_right /= rudder_ho_left_fcs_only AND
    rudder_ho_right /= rudder_ho_right_fcs_only AND
    rudder_ho_right /= indeterminate AND
rudder_ho_left_ah_only /= rudder_ho_right_ah_only AND
    rudder_ho_left_ah_only /= rudder_ho_left_fcs_only AND
    rudder_ho_left_ah_only /= rudder_ho_right_fcs_only AND
    rudder_ho_left_ah_only /= indeterminate AND
rudder_ho_right_ah_only /= rudder_ho_left_fcs_only AND
    rudder_ho_right_ah_only /= rudder_ho_right_fcs_only AND
    rudder_ho_right_ah_only /= indeterminate AND
rudder_ho_left_fcs_only /= rudder_ho_right_fcs_only AND
    rudder_ho_left_fcs_only /= indeterminate AND
rudder_ho_right_fcs_only /= indeterminate

% set of specification levels
proto_speclvl: set[speclvl] =
    {s: speclvl | s = full_service OR s = alt_hold_only OR s = fcs_only OR
    s = rudder_ho_left OR s = rudder_ho_right OR s = rudder_ho_left_ah_only OR
    s = rudder_ho_right_ah_only OR s = rudder_ho_left_fcs_only OR
    s = rudder_ho_right_fcs_only}

% application configurations in the system configurations
proto_sys_configs: sys_configs(proto_speclvl, proto_apps) =
    (LAMBDA (sv: (proto_speclvl)) :
        IF sv = full_service THEN
            (LAMBDA (app: app_spec) :
                IF app = sensors_app THEN sensors_full
                ELSIF app = pilot_interface_app THEN interface_standard
                ELSIF app = autopilot_app THEN ap_standard
                ELSE fcs_standard
                ENDIF)
        ELSIF sv = alt_hold_only THEN
            (LAMBDA (app: app_spec) :
                IF app = sensors_app THEN sensors_full

```

```

        ELSIF app = pilot_interface_app THEN interface_standard
        ELSIF app = autopilot_app THEN ap_ah_only
        ELSE fcs_standard
        ENDIF)
ELSIF sv = rudder_ho_left THEN
    (LAMBDA (app: app_spec) :
        IF app = sensors_app THEN sensors_full
        ELSIF app = pilot_interface_app THEN interface_standard
        ELSIF app = autopilot_app THEN ap_standard
        ELSE fcs_rudder_ho_left
        ENDIF)
ELSIF sv = rudder_ho_right THEN
    (LAMBDA (app: app_spec) :
        IF app = sensors_app THEN sensors_full
        ELSIF app = pilot_interface_app THEN interface_standard
        ELSIF app = autopilot_app THEN ap_standard
        ELSE fcs_rudder_ho_right
        ENDIF)
ELSIF sv = rudder_ho_left_ah_only THEN
    (LAMBDA (app: app_spec) :
        IF app = sensors_app THEN sensors_full
        ELSIF app = pilot_interface_app THEN interface_standard
        ELSIF app = autopilot_app THEN ap_ah_only
        ELSE fcs_rudder_ho_left
        ENDIF)
ELSIF sv = rudder_ho_right_ah_only THEN
    (LAMBDA (app: app_spec) :
        IF app = sensors_app THEN sensors_full
        ELSIF app = pilot_interface_app THEN interface_standard
        ELSIF app = autopilot_app THEN ap_ah_only
        ELSE fcs_rudder_ho_right
        ENDIF)
ELSIF sv = rudder_ho_left_fcs_only THEN
    (LAMBDA (app: app_spec) :
        IF app = sensors_app THEN sensors_full
        ELSIF app = pilot_interface_app THEN interface_standard
        ELSIF app = autopilot_app THEN ap_off
        ELSE fcs_rudder_ho_left
        ENDIF)
ELSIF sv = rudder_ho_right_fcs_only THEN
    (LAMBDA (app: app_spec) :
        IF app = sensors_app THEN sensors_full
        ELSIF app = pilot_interface_app THEN interface_standard
        ELSIF app = autopilot_app THEN ap_off
        ELSE fcs_rudder_ho_right
        ENDIF)
ELSE %fcs_only
    (LAMBDA (app: app_spec) :
        IF app = sensors_app THEN sensors_full
        ELSIF app = pilot_interface_app THEN interface_standard
        ELSIF app = autopilot_app THEN ap_off
        ELSE fcs_standard
        ENDIF)
ENDIF)

```

2. System Transitions

```

% only allow degradation, no repair
degraded(source, target: (proto_speclvl)) : bool =
  IF source = full_service THEN source /= target
  ELSIF source = alt_hold_only THEN
    (target /= full_service AND
     (target = rudder_ho_left_ah_only OR target = rudder_ho_right_ah_only OR
      target = rudder_ho_left_fcs_only OR target =
rudder_ho_right_fcs_only))
  ELSIF source = fcs_only THEN
    (target = rudder_ho_left_fcs_only OR target = rudder_ho_right_fcs_only)
  ELSIF source = rudder_ho_left THEN
    (target = rudder_ho_left_ah_only OR target = rudder_ho_left_fcs_only)
  ELSIF source = rudder_ho_right THEN
    (target = rudder_ho_right_ah_only OR target = rudder_ho_right_fcs_only)
  ELSIF source = rudder_ho_left_ah_only THEN target = rudder_ho_left_fcs_only
  ELSIF source = rudder_ho_right_ah_only THEN target = rudder_ho_right_fcs_only
  ELSE false
  ENDIF

% matches the operating environment at the time
appropriate(target: (proto_speclvl), env: (proto_envs)) : bool =
  IF env(rudder) = hard_over_left THEN
    (target = rudder_ho_left OR target = rudder_ho_left_ah_only OR
     target = rudder_ho_left_fcs_only)
  ELSIF env(rudder) = hard_over_right THEN
    (target = rudder_ho_right OR target = rudder_ho_right_ah_only OR
     target = rudder_ho_right_fcs_only)
  ELSE %rudder working
    (target /= rudder_ho_left AND target /= rudder_ho_left_ah_only AND
     target /= rudder_ho_left_fcs_only AND target /= rudder_ho_right AND
     target /= rudder_ho_right_ah_only AND target /= rudder_ho_right_fcs_only)
  ENDIF
AND
IF env(autopilot) = alt_hold_only THEN
  (target = alt_hold_only OR target = fcs_only OR
   target = rudder_ho_left_ah_only OR target = rudder_ho_left_fcs_only OR
   target = rudder_ho_right_ah_only OR target = rudder_ho_right_fcs_only)
ELSIF env(autopilot) = disabled THEN
  (target = fcs_only OR target = rudder_ho_left_fcs_only OR
   target = rudder_ho_right_fcs_only)
ELSE % autopilot is working
  true
ENDIF
AND
IF env(electrics) = battery THEN
  (target = fcs_only OR target = rudder_ho_left_fcs_only OR
   target = rudder_ho_right_fcs_only)
ELSE % running on alternator
  true
ENDIF
AND
IF (env(electrics) = alternator AND env(autopilot) = fullsvc) THEN
  (target /= alt_hold_only AND target /= fcs_only AND
   target /= rudder_ho_left_ah_only AND target /= rudder_ho_left_fcs_only AND
   target /= rudder_ho_right_ah_only AND target /= rudder_ho_right_fcs_only)
ELSIF (env(electrics) = alternator AND env(autopilot) = alt_hold_only) THEN

```

```

        (target /= fcs_only AND target /= rudder_ho_left_fcs_only AND
         target /= rudder_ho_right_fcs_only)
    ELSE true
    ENDIF

% set of transitions that satisfy the above predicates
proto_trans: set[transition(proto_apps, proto_speclvl, proto_valid_env,
    proto_reachable_env)] =
    {t: transition(proto_apps, proto_speclvl, proto_valid_env,
        proto_reachable_env) |
        (degraded(t`source, t`target) OR t`source = t`target) AND
        appropriate(t`target, t`trigger)}

% axiom to create a mapping that ignores nonexistent env_ids
% when testing for equality
equal_trans: AXIOM
    FORALL (v: valid_env, f1, f2: env(v)) :
        f1(electrics) = f2(electrics) AND f1(autopilot) = f2(autopilot) AND
        f1(rudder) = f2(rudder) => f1 = f2

% axiom to define equality for specification transitions
equal_spec_trans: AXIOM
    FORALL (t1, t2: (proto_trans)) :
        (t1`source = t2`source AND t1`target = t2`target AND
         t1`trigger = t2`trigger) => t1 = t2

% hack
different_env_ids: AXIOM
    electrics /= autopilot AND autopilot /= rudder AND electrics /= rudder

% lemmas to help with TCCs
cov_txn_1: LEMMA
    % Have to cover all transitions out of any start state
    FORALL (e: {e: env(proto_valid_env) |
        e(electrics) = alternator AND e(autopilot) = fullsvc AND
        e(rudder) = working},
        t: (proto_reachable_env`txns)) :
        t`source = e =>
            EXISTS (txn: (proto_trans)) :
                txn`source = full_service AND txn`trigger = t`target

cov_txn_2: LEMMA
    % include start state here so that there can be self-transitions from it
    FORALL (source: (proto_speclvl), t_s: (proto_trans),
        d: (proto_reachable_env`D)) :
        t_s`target = source AND t_s`trigger = d =>
            (FORALL (t_e: (proto_reachable_env`txns)) : t_e`source = d =>
                EXISTS (t_t: (proto_trans)) : t_t`trigger = t_e`target)

cov_txn_3: LEMMA
    % transitions have to be deterministic
    FORALL (source: (proto_speclvl), trigger: (proto_reachable_env`D)) :
        NOT EXISTS (t1, t2: (proto_trans)) :
            t1 /= t2 AND t1`source = source AND t1`trigger = trigger AND
            t2`source = source AND t2`trigger = trigger

```

3. SCRAM table

```

% System configuration information
proto_SCRAM_table: SCRAM_table(proto_apps, proto_speclvl,
                               proto_valid_env, proto_reachable_env) =
(# `configs := proto_sys_configs,
 `primary := full_service,
 `safe := {s: (proto_speclvl) | s = fcs_only},
 `start_env := {e: env(proto_valid_env) |
                e(electrics) = alternator AND e(autopilot) = fullsvc AND
                e(rudder) = working},
 `txns := proto_trans
#)

```

4. Reconfiguration Specification

```

% Function to choose the target specification of a transition
prototype_choose(sv: (proto_speclvl),
                env: env(proto_valid_env)) : (proto_speclvl) =
  IF (EXISTS (target: (proto_speclvl)) :
      EXISTS (t: (proto_SCRAM_table`txns)) :
          t`source = sv AND t`trigger = env AND t`target = target)
  THEN choose({target: (proto_speclvl) |
              EXISTS (t: (proto_SCRAM_table`txns)) :
                  t`source = sv AND t`trigger = env AND t`target = target})
  ELSE sv
  ENDIF

% Time allowed for a prototype reconfiguration
proto_t: real_time = 4*cycle_time

% Complete system specification structure
prototype_reconf_spec: reconf_spec =
(# `apps := proto_apps,
 `app_seq :=
    list2finseq((: sensors_app, pilot_interface_app,
                 autopilot_app, fcs_app :)),
 `S := proto_speclvl,
 `E := proto_valid_env,
 `R := proto_reachable_env,
 `SCRAM_info := proto_SCRAM_table,
 `choose := prototype_choose,
 `T := (LAMBDA (s1, s2: (proto_speclvl)) : proto_t)
#)

END prototype_reconf_spec

```

Appendix C

UAV System Proof Obligations



This appendix contains the TCCs generated for the flight control system specification, and for the prototype reconfiguration system specification. Any TCC that was not actually generated (because it was subsumed, or automatically simplified to `TRUE`) is not included. All of the generated TCCs for the example, along with their proofs, can be found elsewhere [53].

1. FCS Functionality

```
% Disjointness TCC generated (at line 20, column 2) for
% COND st(ap_fcs_cmd) = climb -> st WITH [calc_elevator := e_up],
%   st(ap_fcs_cmd) = hold_alt ->
%     st WITH [calc_elevator := e_neutral],
%   st(ap_fcs_cmd) = descend -> st WITH [calc_elevator := e_down],
%   st(ap_fcs_cmd) = left ->
%     st
%       WITH [calc_left_aileron := a_mid_up,
%             calc_right_aileron := a_mid_down,
%             calc_rudder := r_left],
%   st(ap_fcs_cmd) = hold_hdg ->
%     st
%       WITH [calc_left_aileron := a_neutral,
%             calc_right_aileron := a_neutral,
%             calc_rudder := r_neutral],
%   st(ap_fcs_cmd) = right ->
%     st
%       WITH [calc_left_aileron := a_mid_down,
%             calc_right_aileron := a_mid_up,
%             calc_rudder := r_right],
%   ELSE -> st
% ENDCOND
% proved - complete
fcs_calc_execute_standard_TCC1: OBLIGATION
FORALL (st: data_state):
  NOT (st(ap_fcs_cmd) = climb AND st(ap_fcs_cmd) = hold_alt)
  AND NOT (st(ap_fcs_cmd) = climb AND st(ap_fcs_cmd) = descend)
  AND NOT (st(ap_fcs_cmd) = climb AND st(ap_fcs_cmd) = left)
  AND NOT (st(ap_fcs_cmd) = climb AND st(ap_fcs_cmd) = hold_hdg)
  AND NOT (st(ap_fcs_cmd) = climb AND st(ap_fcs_cmd) = right)
  AND NOT (st(ap_fcs_cmd) = hold_alt AND st(ap_fcs_cmd) = descend)
  AND NOT (st(ap_fcs_cmd) = hold_alt AND st(ap_fcs_cmd) = left)
  AND NOT (st(ap_fcs_cmd) = hold_alt AND st(ap_fcs_cmd) = hold_hdg)
  AND NOT (st(ap_fcs_cmd) = hold_alt AND st(ap_fcs_cmd) = right)
  AND NOT (st(ap_fcs_cmd) = descend AND st(ap_fcs_cmd) = left)
  AND NOT (st(ap_fcs_cmd) = descend AND st(ap_fcs_cmd) = hold_hdg)
  AND NOT (st(ap_fcs_cmd) = descend AND st(ap_fcs_cmd) = right)
  AND NOT (st(ap_fcs_cmd) = left AND st(ap_fcs_cmd) = hold_hdg)
  AND NOT (st(ap_fcs_cmd) = left AND st(ap_fcs_cmd) = right)
  AND NOT (st(ap_fcs_cmd) = hold_hdg AND st(ap_fcs_cmd) = right);
```

2. FCS

```
% Subtype TCC generated (at line 26, column 42) for
```

```

    % (LAMBDA (s: data_state): TRUE)
    % expected type predicate(fcs_calc_scope)
    % proved - complete
fcs_calc_TCC1: OBLIGATION
  FORALL (sv: (fcs_calc_svcs)):
    FORALL (st: data_state): FORALL (st2: data_state): TRUE;

% Subtype TCC generated (at line 50, column 44) for
  % (LAMBDA (s: data_state): TRUE)
  % expected type predicate(fcs_adjust_scope)
  % proved - complete
fcs_adjust_TCC1: OBLIGATION
  FORALL (sv: (fcs_adjust_svcs)):
    FORALL (st: data_state): FORALL (st2: data_state): TRUE;

% Subtype TCC generated (at line 72, column 44) for
  % (LAMBDA (s: data_state): TRUE)
  % expected type predicate(fcs_output_scope)
  % proved - complete
fcs_output_TCC1: OBLIGATION
  FORALL (sv: (fcs_output_svcs)):
    FORALL (st: data_state): FORALL (st2: data_state): TRUE;

% Subtype TCC generated (at line 79, column 1) for
  % {m: module_spec | m = fcs_calc OR m = fcs_adjust OR m = fcs_output}
  % expected type finite_set[module_spec]
  % proved - complete
fcs_modules_TCC1: OBLIGATION
  is_finite[module_spec]
    ({m: module_spec | m = fcs_calc OR m = fcs_adjust OR m = fcs_output});

% Subtype TCC generated (at line 83, column 2) for
  % (LAMBDA (m: (fcs_modules))):
  %   IF (m = fcs_calc) THEN fcs_calc_standard
  %   ELSIF (m = fcs_adjust) THEN fcs_adjust_nc
  %   ELSE fcs_output_standard
  %   ENDIF)
  % expected type service_map(fcs_modules)
  % proved - complete
fcs_standard_svcmap_TCC1: OBLIGATION
  FORALL (mod: (fcs_modules)):
    mod`sv
      (IF (mod = fcs_calc) THEN fcs_calc_standard
        ELSIF (mod = fcs_adjust) THEN fcs_adjust_nc
        ELSE fcs_output_standard
        ENDIF);

% Subtype TCC generated (at line 89, column 2) for
  % (LAMBDA (m: (fcs_modules))):
  %   IF (m = fcs_calc) THEN fcs_calc_standard
  %   ELSIF (m = fcs_adjust) THEN fcs_adjust_ho_left
  %   ELSE fcs_output_ho
  %   ENDIF)
  % expected type service_map(fcs_modules)
  % proved - complete
fcs_ho_left_svcmap_TCC1: OBLIGATION
  FORALL (mod: (fcs_modules)):
    mod`sv
      (IF (mod = fcs_calc) THEN fcs_calc_standard

```

```

        ELSIF (mod = fcs_adjust) THEN fcs_adjust_ho_left
        ELSE fcs_output_ho
        ENDIF);

% Subtype TCC generated (at line 95, column 2) for
% (LAMBDA (m: (fcs_modules))):
%   IF (m = fcs_calc) THEN fcs_calc_standard
%   ELSIF (m = fcs_adjust) THEN fcs_adjust_ho_right
%   ELSE fcs_output_ho
%   ENDIF)
% expected type  service_map(fcs_modules)
% proved - complete
fcs_ho_right_svcmap_TCC1: OBLIGATION
FORALL (mod: (fcs_modules)):
  mod`sv
  (IF (mod = fcs_calc) THEN fcs_calc_standard
  ELSIF (mod = fcs_adjust) THEN fcs_adjust_ho_right
  ELSE fcs_output_ho
  ENDIF);

% Subtype TCC generated (at line 115, column 37) for  fcs_calc
% expected type  (fcs_modules)
% proved - complete
fcs_execute_TCC1: OBLIGATION FORALL (s: (fcs_svcs)): fcs_modules(fcs_calc);

% Subtype TCC generated (at line 115, column 23) for
% fcs_svcmap(s)(fcs_calc)
% expected type  (fcs_calc_svcs)
% proved - complete
fcs_execute_TCC2: OBLIGATION
FORALL (s: (fcs_svcs)): fcs_calc_svcs(fcs_svcmap(s)(fcs_calc));

% Subtype TCC generated (at line 114, column 39) for  fcs_adjust
% expected type  (fcs_modules)
% proved - complete
fcs_execute_TCC3: OBLIGATION FORALL (s: (fcs_svcs)): fcs_modules(fcs_adjust);

% Subtype TCC generated (at line 114, column 25) for
% fcs_svcmap(s)(fcs_adjust)
% expected type  (fcs_adjust_svcs)
% proved - complete
fcs_execute_TCC4: OBLIGATION
FORALL (s: (fcs_svcs)): fcs_adjust_svcs(fcs_svcmap(s)(fcs_adjust));

% Subtype TCC generated (at line 113, column 38) for  fcs_output
% expected type  (fcs_modules)
% proved - complete
fcs_execute_TCC5: OBLIGATION FORALL (s: (fcs_svcs)): fcs_modules(fcs_output);

% Subtype TCC generated (at line 113, column 24) for
% fcs_svcmap(s)(fcs_output)
% expected type  (fcs_output_svcs)
% proved - complete
fcs_execute_TCC6: OBLIGATION
FORALL (s: (fcs_svcs)): fcs_output_svcs(fcs_svcmap(s)(fcs_output));

% Subtype TCC generated (at line 167, column 13) for  fcs_execute
% expected type  [(fcs_svcs) -> func(app_scope(fcs_modules))]
% proved - complete

```

```

fcs_app_TCC1: OBLIGATION
  FORALL (x1: (fcs_svcs)):
    (FORALL (st: data_state):
      (fcs_execute(x1)`pre(st) =>
        (FORALL (st2: data_state):
          (FORALL (id: (app_scope(fcs_modules))): st2(id) = st(id)) =>
            fcs_execute(x1)`pre(st2))))
    AND
    (FORALL (d: data_state, id: data_id):
      NOT app_scope(fcs_modules)(id) => fcs_execute(x1)`f(d)(id) = d(id));

% Subtype TCC generated (at line 168, column 15) for fcs_exec_halt
% expected type [(fcs_svcs) -> func(app_scope(fcs_modules))]
% proved - complete
fcs_app_TCC2: OBLIGATION
  FORALL (x1: (fcs_svcs)):
    (FORALL (st: data_state):
      (fcs_exec_halt(x1)`pre(st) =>
        (FORALL (st2: data_state):
          (FORALL (id: (app_scope(fcs_modules))): st2(id) = st(id)) =>
            fcs_exec_halt(x1)`pre(st2))))
    AND
    (FORALL (d: data_state, id: data_id):
      NOT app_scope(fcs_modules)(id) => fcs_exec_halt(x1)`f(d)(id) = d(id));

% Subtype TCC generated (at line 170, column 10) for fcs_halt
% expected type [(fcs_svcs) -> func(app_scope(fcs_modules))]
% proved - complete
fcs_app_TCC3: OBLIGATION
  FORALL (x1: (fcs_svcs)):
    (FORALL (st: data_state):
      (fcs_halt(x1)`pre(st) =>
        (FORALL (st2: data_state):
          (FORALL (id: (app_scope(fcs_modules))): st2(id) = st(id)) =>
            fcs_halt(x1)`pre(st2))))
    AND
    (FORALL (d: data_state, id: data_id):
      NOT app_scope(fcs_modules)(id) => fcs_halt(x1)`f(d)(id) = d(id));

% Subtype TCC generated (at line 169, column 10) for fcs_prep
% expected type [(fcs_svcs) -> func(app_scope(fcs_modules))]
% proved - complete
fcs_app_TCC4: OBLIGATION
  FORALL (x1: (fcs_svcs)):
    (FORALL (st: data_state):
      (fcs_prep(x1)`pre(st) =>
        (FORALL (st2: data_state):
          (FORALL (id: (app_scope(fcs_modules))): st2(id) = st(id)) =>
            fcs_prep(x1)`pre(st2))))
    AND
    (FORALL (d: data_state, id: data_id):
      NOT app_scope(fcs_modules)(id) => fcs_prep(x1)`f(d)(id) = d(id));

```

3. Example Reconfiguration Specification

```

% Subtype TCC generated (at line 93, column 4) for

```

```

% (restrict[app_spec, (proto_apps), app_svclvl]
%   ((LAMBDA (app: app_spec):
%     IF app = sensors_app THEN sensors_full
%     ELSIF app = pilot_interface_app THEN interface_standard
%     ELSIF app = autopilot_app THEN ap_standard
%     ELSE fcs_standard
%     ENDIF)))
% expected type  valid_app_svcs(proto_apps)
% proved - complete
proto_sys_configs_TCC1: OBLIGATION
FORALL (sv: (proto_speclvl)):
sv = full_service IMPLIES
(FORALL (app_1: (proto_apps)):
app_1`svcs
(restrict[app_spec, (proto_apps), app_svclvl]
((LAMBDA (app: app_spec):
IF app = sensors_app THEN sensors_full
ELSIF app = pilot_interface_app THEN interface_standard
ELSIF app = autopilot_app THEN ap_standard
ELSE fcs_standard
ENDIF))
(app_1)));

% Subtype TCC generated (at line 100, column 4) for
% (restrict[app_spec, (proto_apps), app_svclvl]
%   ((LAMBDA (app: app_spec):
%     IF app = sensors_app THEN sensors_full
%     ELSIF app = pilot_interface_app THEN interface_standard
%     ELSIF app = autopilot_app THEN ap_ah_only
%     ELSE fcs_standard
%     ENDIF)))
% expected type  valid_app_svcs(proto_apps)
% proved - complete
proto_sys_configs_TCC2: OBLIGATION
FORALL (sv: (proto_speclvl)):
NOT sv = full_service AND sv = alt_hold_only IMPLIES
(FORALL (app_1: (proto_apps)):
app_1`svcs
(restrict[app_spec, (proto_apps), app_svclvl]
((LAMBDA (app: app_spec):
IF app = sensors_app THEN sensors_full
ELSIF app = pilot_interface_app THEN interface_standard
ELSIF app = autopilot_app THEN ap_ah_only
ELSE fcs_standard
ENDIF))
(app_1)));

% Subtype TCC generated (at line 107, column 4) for
% (restrict[app_spec, (proto_apps), app_svclvl]
%   ((LAMBDA (app: app_spec):
%     IF app = sensors_app THEN sensors_full
%     ELSIF app = pilot_interface_app THEN interface_standard
%     ELSIF app = autopilot_app THEN ap_standard
%     ELSE fcs_rudder_ho_left
%     ENDIF)))
% expected type  valid_app_svcs(proto_apps)
% proved - complete
proto_sys_configs_TCC3: OBLIGATION
FORALL (sv: (proto_speclvl)):

```

```

NOT sv = full_service AND NOT sv = alt_hold_only AND sv = rudder_ho_left
IMPLIES
(FORALL (app_1: (proto_apps)):
  app_1`svcs
    (restrict[app_spec, (proto_apps), app_svclvl]
      ((LAMBDA (app: app_spec):
        IF app = sensors_app THEN sensors_full
        ELSIF app = pilot_interface_app THEN interface_standard
        ELSIF app = autopilot_app THEN ap_standard
        ELSE fcs_rudder_ho_left
        ENDIF))
      (app_1)));

% Subtype TCC generated (at line 114, column 4) for
% (restrict[app_spec, (proto_apps), app_svclvl]
% ((LAMBDA (app: app_spec):
%   IF app = sensors_app THEN sensors_full
%   ELSIF app = pilot_interface_app THEN interface_standard
%   ELSIF app = autopilot_app THEN ap_standard
%   ELSE fcs_rudder_ho_right
%   ENDIF)))
% expected type valid_app_svcs(proto_apps)
% proved - complete
proto_sys_configs_TCC4: OBLIGATION
FORALL (sv: (proto_speclvl)):
  NOT sv = full_service AND
  NOT sv = alt_hold_only AND
  NOT sv = rudder_ho_left AND sv = rudder_ho_right
  IMPLIES
  (FORALL (app_1: (proto_apps)):
    app_1`svcs
      (restrict[app_spec, (proto_apps), app_svclvl]
        ((LAMBDA (app: app_spec):
          IF app = sensors_app THEN sensors_full
          ELSIF app = pilot_interface_app THEN interface_standard
          ELSIF app = autopilot_app THEN ap_standard
          ELSE fcs_rudder_ho_right
          ENDIF))
        (app_1)));

% Subtype TCC generated (at line 121, column 4) for
% (restrict[app_spec, (proto_apps), app_svclvl]
% ((LAMBDA (app: app_spec):
%   IF app = sensors_app THEN sensors_full
%   ELSIF app = pilot_interface_app THEN interface_standard
%   ELSIF app = autopilot_app THEN ap_ah_only
%   ELSE fcs_rudder_ho_left
%   ENDIF)))
% expected type valid_app_svcs(proto_apps)
% proved - complete
proto_sys_configs_TCC5: OBLIGATION
FORALL (sv: (proto_speclvl)):
  NOT sv = full_service AND NOT sv = alt_hold_only
  AND NOT sv = rudder_ho_left AND NOT sv = rudder_ho_right
  AND sv = rudder_ho_left_ah_only
  IMPLIES
  (FORALL (app_1: (proto_apps)):
    app_1`svcs
      (restrict[app_spec, (proto_apps), app_svclvl]

```

```

        ((LAMBDA (app: app_spec):
          IF app = sensors_app THEN sensors_full
          ELSIF app = pilot_interface_app THEN interface_standard
          ELSIF app = autopilot_app THEN ap_ah_only
          ELSE fcs_rudder_ho_left
          ENDIF))
      (app_1));

% Subtype TCC generated (at line 128, column 4) for
% (restrict[app_spec, (proto_apps), app_svclvl]
%   ((LAMBDA (app: app_spec):
%     IF app = sensors_app THEN sensors_full
%     ELSIF app = pilot_interface_app THEN interface_standard
%     ELSIF app = autopilot_app THEN ap_ah_only
%     ELSE fcs_rudder_ho_right
%     ENDIF)))
% expected type valid_app_svcs(proto_apps)
% proved - complete
proto_sys_configs_TCC6: OBLIGATION
FORALL (sv: (proto_speclvl)):
  NOT sv = full_service AND NOT sv = alt_hold_only
  AND NOT sv = rudder_ho_left AND NOT sv = rudder_ho_right
  AND NOT sv = rudder_ho_left_ah_only AND sv = rudder_ho_right_ah_only
  IMPLIES
  (FORALL (app_1: (proto_apps)):
    app_1`svcs
    (restrict[app_spec, (proto_apps), app_svclvl]
      ((LAMBDA (app: app_spec):
        IF app = sensors_app THEN sensors_full
        ELSIF app = pilot_interface_app THEN interface_standard
        ELSIF app = autopilot_app THEN ap_ah_only
        ELSE fcs_rudder_ho_right
        ENDIF))
      (app_1)));

% Subtype TCC generated (at line 135, column 4) for
% (restrict[app_spec, (proto_apps), app_svclvl]
%   ((LAMBDA (app: app_spec):
%     IF app = sensors_app THEN sensors_full
%     ELSIF app = pilot_interface_app THEN interface_standard
%     ELSIF app = autopilot_app THEN ap_off
%     ELSE fcs_rudder_ho_left
%     ENDIF)))
% expected type valid_app_svcs(proto_apps)
% proved - complete
proto_sys_configs_TCC7: OBLIGATION
FORALL (sv: (proto_speclvl)):
  NOT sv = full_service AND NOT sv = alt_hold_only
  AND NOT sv = rudder_ho_left AND NOT sv = rudder_ho_right
  AND NOT sv = rudder_ho_left_ah_only AND NOT sv = rudder_ho_right_ah_only
  AND sv = rudder_ho_left_fcs_only
  IMPLIES
  (FORALL (app_1: (proto_apps)):
    app_1`svcs
    (restrict[app_spec, (proto_apps), app_svclvl]
      ((LAMBDA (app: app_spec):
        IF app = sensors_app THEN sensors_full
        ELSIF app = pilot_interface_app THEN interface_standard
        ELSIF app = autopilot_app THEN ap_off

```

```

        ELSE fcs_rudder_ho_left
        ENDIF))
    (app_1));

% Subtype TCC generated (at line 142, column 4) for
% (restrict[app_spec, (proto_apps), app_svclvl]
% ((LAMBDA (app: app_spec):
%     IF app = sensors_app THEN sensors_full
%     ELSIF app = pilot_interface_app THEN interface_standard
%     ELSIF app = autopilot_app THEN ap_off
%     ELSE fcs_rudder_ho_right
%     ENDIF)))
% expected type valid_app_svcs(proto_apps)
% proved - complete
proto_sys_configs_TCC8: OBLIGATION
FORALL (sv: (proto_speclvl)):
    NOT sv = full_service AND NOT sv = alt_hold_only
    AND NOT sv = rudder_ho_left AND NOT sv = rudder_ho_right
    AND NOT sv = rudder_ho_left_ah_only AND NOT sv = rudder_ho_right_ah_only
    AND NOT sv = rudder_ho_left_fcs_only AND sv = rudder_ho_right_fcs_only
    IMPLIES
    (FORALL (app_1: (proto_apps)):
        app_1`svcs
        (restrict[app_spec, (proto_apps), app_svclvl]
        ((LAMBDA (app: app_spec):
            IF app = sensors_app THEN sensors_full
            ELSIF app = pilot_interface_app THEN interface_standard
            ELSIF app = autopilot_app THEN ap_off
            ELSE fcs_rudder_ho_right
            ENDIF))
        (app_1)));

% Subtype TCC generated (at line 149, column 4) for
% (restrict[app_spec, (proto_apps), app_svclvl]
% ((LAMBDA (app: app_spec):
%     IF app = sensors_app THEN sensors_full
%     ELSIF app = pilot_interface_app THEN interface_standard
%     ELSIF app = autopilot_app THEN ap_off
%     ELSE fcs_standard
%     ENDIF)))
% expected type valid_app_svcs(proto_apps)
% proved - complete
proto_sys_configs_TCC9: OBLIGATION
FORALL (sv: (proto_speclvl)):
    NOT sv = full_service AND NOT sv = alt_hold_only
    AND NOT sv = rudder_ho_left AND NOT sv = rudder_ho_right
    AND NOT sv = rudder_ho_left_ah_only AND NOT sv = rudder_ho_right_ah_only
    AND NOT sv = rudder_ho_left_fcs_only AND NOT sv = rudder_ho_right_fcs_only
    IMPLIES
    (FORALL (app_1: (proto_apps)):
        app_1`svcs
        (restrict[app_spec, (proto_apps), app_svclvl]
        ((LAMBDA (app: app_spec):
            IF app = sensors_app THEN sensors_full
            ELSIF app = pilot_interface_app THEN interface_standard
            ELSIF app = autopilot_app THEN ap_off
            ELSE fcs_standard
            ENDIF))
        (app_1)));

```

```

% Subtype TCC generated (at line 222, column 24) for t`trigger
% expected type (proto_envs)
% proved - complete
proto_trans_TCC1: OBLIGATION
FORALL (t:
    transition(proto_apps, proto_speclvl, proto_valid_env,
                proto_reachable_env)):
    (degraded(t`source, t`target) OR t`source = t`target) IMPLIES
    proto_envs(t`trigger);

% Subtype TCC generated (at line 268, column 13) for full_service
% expected type (proto_speclvl)
% proved - complete
proto_SCRAM_table_TCC1: OBLIGATION proto_speclvl(full_service);

% Subtype TCC generated (at line 280, column 20) for
% {target: (proto_speclvl) |
%   EXISTS (t: (proto_SCRAM_table`txns)):
%     t`source = sv AND t`trigger = env AND t`target = target}
% expected type p: (nonempty?[(proto_speclvl)])
% proved - complete
prototype_choose_TCC1: OBLIGATION
FORALL (env: env(proto_valid_env), sv: (proto_speclvl)):
    (EXISTS (target: (proto_speclvl)):
        EXISTS (t: (proto_SCRAM_table`txns)):
            t`source = sv AND t`trigger = env AND t`target = target)
    IMPLIES
    nonempty?[(proto_speclvl)
        ({target: (proto_speclvl) |
            EXISTS (t: (proto_SCRAM_table`txns)):
                t`source = sv AND t`trigger = env AND t`target = target})];

% Subtype TCC generated (at line 293, column 2) for
% list2finseq((: sensors_app, pilot_interface_app, autopilot_app,
%             fcs_app :))
% expected type finseq[(proto_apps)]
% proved - complete
prototype_reconf_spec_TCC1: OBLIGATION
FORALL (x1:
    below[list2finseq[app_spec]
        ((: sensors_app, pilot_interface_app, autopilot_app,
            fcs_app :))`length]):
    proto_apps(list2finseq[app_spec]
        ((: sensors_app, pilot_interface_app, autopilot_app,
            fcs_app :))`seq
        (x1));

% 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 - complete
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);

% Subtype TCC generated (at line 298, column 12) for prototype_choose
% expected type [(restrict[speclvl,
%           {sp: speclvl | NOT sp = indeterminate},
%           boolean]
%           (proto_speclvl)),
% env(proto_valid_env)] ->
%           (restrict[speclvl,
%           {sp: speclvl | NOT sp = indeterminate},
%           boolean]
%           (proto_speclvl))]
% proved - complete
prototype_reconf_spec_TCC3: OBLIGATION

```

```

(FORALL (x: speclvl):
  proto_speclvl(x) IFF
  NOT x = indeterminate AND
  restrict[speclvl, {sp: speclvl | NOT sp = indeterminate}, boolean]
    (proto_speclvl)(x))
AND
(FORALL (x1: [(proto_speclvl), env(proto_valid_env)]):
  NOT prototype_choose(x1) = indeterminate AND
  restrict[speclvl, {sp: speclvl | NOT sp = indeterminate}, boolean]
    (proto_speclvl)(prototype_choose(x1)));

% Subtype TCC generated (at line 299, column 43) for proto_t
% expected type {t: real_time | t >= 4 * cycle_time}
% proved - complete
prototype_reconf_spec_TCC4: OBLIGATION
FORALL (s1, s2: (proto_speclvl)): proto_t >= 4 * cycle_time;

% Subtype TCC generated (at line 299, column 8) for
% (LAMBDA (s1, s2: (proto_speclvl)): proto_t)
% expected type [[(restrict[speclvl,
%                 {sp: speclvl | NOT sp = indeterminate},
%                 boolean]
%                 (proto_speclvl)),
%                 (restrict[speclvl,
%                 {sp: speclvl | NOT sp = indeterminate},
%                 boolean]
%                 (proto_speclvl)))] ->
%                 {t: real_time | t >= 4 * cycle_time}]
% proved - complete
prototype_reconf_spec_TCC5: OBLIGATION
(FORALL (x: speclvl):
  proto_speclvl(x) IFF
  NOT x = indeterminate AND
  restrict[speclvl, {sp: speclvl | NOT sp = indeterminate}, boolean]
    (proto_speclvl)(x))
AND
(FORALL (x: speclvl):
  proto_speclvl(x) IFF
  NOT x = indeterminate AND
  restrict[speclvl, {sp: speclvl | NOT sp = indeterminate}, boolean]
    (proto_speclvl)(x));

```

Appendix D

Example Property Proof



This appendix contains the proof script for:

```

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`svcmmap(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`svcmmap(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`svcmmap(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

```

assured_reconfig.CP5: proved - complete [shostak](13048.43 s)

```

(""
  (skosimp)
  (split)
  ("1"
    (lemma "reconf_length")
    (inst -1 "s!1" "r!1")
    (typepred "r!1")
    (typepred "s!1`tr")
    (expand "get_reconfigs")
    (hide -2 -3 -4)
    (flatten)
    (case "r!1`end_c - r!1`start_c = 1")
    ("1"
      (lemma "reconf_halt")
      (expand "reconfig_end?")
      (split -6)
      ("1"
        (expand "reconfig_start?")
        (skosimp)
        (inst -1 "app!1")
        (inst -2 "s!1" "r!1" "app!1")
        (hide -4 -5 -6 -7 -8)
        (grind)
        ("2" (propax)))
      ("2"

```

```

(case "r!1`end_c - r!1`start_c = 2")
(("1"
  (lemma "reconf_prep")
  (expand "reconfig_end?")
  (inst -1 "s!1" "r!1")
  (split -1)
  (("1"
    (split -6)
    (("1"
      (split 3)
      (("1" (propax))
      ("2"
        (skosimp 1)
        (case "r!1`end_c - r!1`start_c = 2")
        (("1"
          (inst -2 "app!1")
          (inst -3 "app!1")
          (hide -5 -6 -7 -8 2 3)
          (split -3)
          (("1" (hide 1 2) (grind))
          ("2"
            (lemma "CP4")
            (inst -1 "s!1" "r!1`start_c+2")
            (split -1)
            (("1"
              (flatten)
              (expand "inv")
              (inst -1 "app!1")
              (expand "inv")
              (split 1)
              (("1" (hide -1 -3 -5 -6 -7 2) (grind))
              ("2"
                (skosimp)
                (inst -1 "m!1")
                (hide -2 -3 -5 -6 -7 2)
                (grind))))
            ("2"
              (flatten)
              (inst -2 "app!1")
              (split)
              (("1"
                (flatten)
                (inst 3 "app!1")
                (hide -2 -3 -4 -5 -6 -8 -9 -10 1 2)
                (grind))
                ("2" (hide -2 -6 -7 -8 2) (grind))))))
            ("3" (inst 2 "app!1") (hide -3 -4 -5 1) (grind))))
            ("2" (propax))))))
          ("2" (propax))))
          ("2" (hide -1 -2 -6 -4 -5 4 5) (assert))))
          ("2" (hide -1 -3 -4 -5 4 5) (assert))))))
    (skosimp)
    (lemma "reconf_prep")
    (inst -1 "s!1" "r!1")
    (split)
    (("1"
      (inst -1 "app!1")
      (split -1)

```

```

(("1"
 (flatten)
 (lemma "reconf_train")
 (inst -1 "s!1" "r!1")
 (split)
 ("1"
  (inst -1 "app!1")
  (hide 3 4)
  (case "r!1`end_c-r!1`start_c < 3")
  ("1"
   (hide -2 -3 1 2)
   (typepred "r!1")
   (expand "get_reconfigs")
   (flatten)
   (case "r!1`end_c - r!1`start_c = 1")
   ("1"
    (expand "reconfig_end?")
    (split -4)
    ("1"
     (lemma "reconf_halt")
     (inst -2 "app!1")
     (inst -1 "s!1" "r!1" "app!1")
     (hide -5 -6 -7 1)
     (grind))
     ("2" (propax))))
    ("2"
     (case "r!1`end_c-r!1`start_c = 2")
     ("1"
      (reveal -2)
      (expand "reconfig_end?")
      (split -5)
      (("1" (inst -1 "app!1") (hide -4 -5 -6 -7 1 2) (grind))
       ("2" (propax))))
      ("2" (hide -2 -3 -4 3) (grind))))))
   ("2"
    (typepred "s!1`tr")
    (inst -1 "r!1`start_c+2")
    (hide -2 -3 -4)
    (expand "system_monitor")
    (lift-if)
    (split)
    ("1"
     (flatten)
     (hide -2)
     (skosimp)
     (reveal -8)
     (inst -1 "app!2")
     (("1" (hide -3 -4 1 2 3 4 5) (grind))
      ("2"
       (hide -1 -2 -3 2 3 4 5 6)
       (typepred "app!2")
       (typepred "s!1`tr(r!1`start_c+2)")
       (grind))))
     ("2"
      (flatten)
      (hide 1)
      (expand "next_config")
      (lift-if)
      (split)

```

```

(("1"
  (flatten)
  (lemma "int_prep_len")
  (inst -1 "s!1" "r!1")
  (split)
  (("1" (hide -2 -3 -4 -5 2 3 4) (assert))
   ("2" (hide -2 -3 -4 2 3 4 5) (grind))
   ("3" (hide -1 -2 -3 -4 3 4 5) (grind))))
("2"
  (split)
  (("1"
    (flatten)
    (split)
    (("1"
      (flatten)
      (skosimp)
      (reveal -7)
      (inst -1 "app!2")
      (("1"
        (hide -3 -4 -5 -6 2 3 4)
        (split)
        (("1" (grind)) ("2" (grind)) ("3" (grind))))
       ("2"
        (hide -1 -2 -3 -4 -5 2 3 4 5)
        (typepred "app!2")
        (typepred "s!1`tr(r!1`start_c+2)")
        (grind))))
      ("2"
        (flatten)
        (split)
        (("1"
          (flatten)
          (hide -1 -2 -5 1 2 5)
          (lemma "reconf_length")
          (inst -1 "s!1" "r!1")
          (grind))
         ("2"
          (flatten)
          (hide -1 -2 -3 2 3 4 5 6)
          (inst 1 "app!1")
          (("1" (grind))
           ("2"
            (typepred "app!1")
            (typepred "s!1`tr(r!1`start_c+2)")
            (grind))))))))
      ("2"
        (flatten)
        (split)
        (("1"
          (flatten)
          (reveal -7)
          (hide -3 -4 -5 1 2 3 4 5)
          (skosimp)
          (inst -1 "app!2")
          (("1" (grind))
           ("2"
            (typepred "s!1`tr(2 + r!1`start_c)")
            (grind))))
         ("2"
          (grind))))
      ("2"
        (grind))))))

```

```

(flatten)
(split)
(("1"
  (flatten)
  (lemma "train_time")
  (inst -1 "s!1" "r!1`start_c+3" "app!1")
  (split)
  ("1"
    (hide -2 -3 -4 1 5)
    (lemma "reconf_length")
    (inst -1 "s!1" "r!1")
    (grind))
  ("2"
    (hide -1 -3 2 3 4 6)
    (lemma "change_to_interrupt_rec")
    (inst -1
      "s!1`tr(2 + r!1`start_c)
      WITH [`reconf_st
        := LAMBDA
          (app:
            (s!1`tr(2 +
r!1`start_c)`sp`apps)):
          IF (s!1`tr(2 +
r!1`start_c)`reconf_st
            (app)
            =
            prepping)
          THEN training
          ELSE normal
          ENDIF]"
      "app!1"
      "card(s!1`tr(2 + r!1`start_c)`sp`apps) - 1")
    ("1"
      (split -1)
      (("1" (hide -2 -3 1 2 3) (grind))
       ("2" (grind))
       ("3"
         (inst 3 "app!1")
         (lemma "reconf_length")
         (inst -1 "s!1" "r!1")
         (grind))))
    ("2"
      (hide -1 -2 2 3 4)
      (lemma "nonempty_apps")
      (inst -1 "s!1`tr(2 + r!1`start_c)")
    ("3"
      (hide -1 -2 2 3 4)
      (typepred "s!1`tr(2 + r!1`start_c)")
      (grind))))))
  ("2"
    (flatten)
    (inst 1 "app!1")
    (("1" (hide -1 -2 2 3 4 5 6 7) (grind))
     ("2"
       (hide -1 -2 -3 2 3 4 5 6 7)
       (typepred "s!1`tr(2 + r!1`start_c)")
       (grind)))))))))
("2"
  (hide -1 2 3 4 5)

```

```

(reveal -3)
(typepred "r!1")
(expand "get_reconfigs")
(flatten)
(reveal -1)
(case "r!1`end_c - r!1`start_c <= 2")
(("1"
  (case "r!1`end_c-r!1`start_c=2")
  ("1"
    (inst -7 "r!1`start_c+2")
    (hide -2 -4 -5)
    (expand "reconfig_end?")
    (split)
    (("1" (inst -1 "app!1") (hide -4 -5 1 2) (grind))
     ("2" (propax))))
  ("2"
    (case "r!1`end_c-r!1`start_c=1")
    ("1"
      (expand "reconfig_end?")
      (lemma "reconf_halt")
      (inst -1 "s!1" "r!1" "app!1")
      (hide -3 -4 -5 -6 -8 -9)
      (split -3)
      (("1" (inst -1 "app!1") (grind)) ("2" (propax))))
      ("2" (hide -2 -4 -5 -6 -7 3 4) (grind))))))
  ("2" (hide -1 -2 -3 -4 -5 -6 3) (grind))))))
("2"
  (hide 1 3)
  (lemma "CP4")
  (inst -1 "s!1" "r!1`start_c+2")
  (case "r!1`end_c-r!1`start_c=2")
  (("1"
    (split -2)
    ("1"
      (expand "inv")
      (inst -1 "app!1")
      (split 1)
      ("1" (hide -1 2) (grind))
      ("2"
        (skosimp)
        (expand "inv")
        (inst -1 "m!1")
        (hide -3 -4 2)
        (grind))))
    ("2"
      (flatten)
      (inst -2 "app!1")
      (split -2)
      (("1" (flatten) (hide -2 -3 -4 -5 -6 1 2 3) (grind))
       ("2" (hide -2 -5 2) (grind))))))
  ("2"
    (reveal -1)
    (hide -2)
    (case "r!1`end_c-r!1`start_c = 1")
    (("1"
      (hide -2)
      (lemma "reconf_halt")
      (inst -1 "s!1" "r!1" "app!1")
      (typepred "r!1")

```

```

(expand "get_reconfigs")
(flatten)
(expand "reconfig_end?")
(split -3)
(("1" (inst -1 "app!1") (hide -2 -3 -4 -7 -8 1 2 3) (grind))
 ("2" (propax)))
("2"
 (case "r!1`end_c-r!1`start_c=3")
 ("1"
  (inst -2 "s!1" "r!1`start_c+3")
  (lemma "reconf_train")
  (inst -1 "s!1" "r!1")
  (split -1)
  ("1"
   (inst -1 "app!1")
   (split 3)
   ("1" (hide -3 2 3 4) (grind))
   ("2"
    (split -3)
    ("1"
     (expand "inv")
     (skosimp)
     (inst -1 "app!1")
     (expand "inv")
     (inst -1 "m!1")
     (hide -2 -4 -5 2 3 4)
     (grind))
    ("2"
     (flatten)
     (inst -2 "app!1")
     (split -2)
     ("1"
      (flatten)
      (inst 5 "app!1")
      (hide -2 -3 -4 -5 -7 -8 1 2 3 4)
      (grind))
     ("2"
      (expand "inv")
      (skosimp)
      (inst -1 "m!1")
      (hide -2 -3 -5 -6 2 3 4)
      (grind))))))
   ("2" (hide -2 -3 2 3 4 5) (grind)))
 ("2"
  (lemma "reconf_length")
  (typepred "r!1")
  (expand "get_reconfigs")
  (flatten)
  (hide -2 -3 -4 -6 -7 -8 4 5)
  (inst -2 "s!1" "r!1")
  (grind))))))
("3"
 (hide 1 2 3)
 (inst 1 "app!1")
 (lemma "int_prep_len")
 (inst -1 "s!1" "r!1")
 (split)
 (("1" (grind)) ("2" (inst 1 "app!1"))
 ("3"

```

```

(case "r!1`end_c-r!1`start_c=1")
(("1"
  (lemma "reconf_halt")
  (inst -1 "s!1" "r!1" "app!1")
  (typepred "r!1")
  (expand "get_reconfigs")
  (flatten)
  (expand "reconfig_end?")
  (hide -2 -4 -7)
  (split)
  (("1" (inst -1 "app!1") (grind))
   ("2" (skosimp) (reveal 1) (inst 1 "app!2"))))
("2"
  (typepred "r!1")
  (expand "get_reconfigs")
  (flatten)
  (hide -2 -3 -4 -5)
  (grind))))))
("2"
 (hide 2 3 4)
 (typepred "r!1")
 (expand "get_reconfigs")
 (flatten)
 (case "r!1`end_c-r!1`start_c=1")
 (("1"
   (expand "reconfig_end?")
   (lemma "reconf_halt")
   (inst -1 "s!1" "r!1" "app!1")
   (hide -4 -6)
   (split -4)
   (("1" (inst -1 "app!1") (grind)) ("2" (propax))))
  ("2" (hide -2 -3 -4 3) (grind))))))

```