

# Assured Reconfiguration of Embedded Real-Time Software

Elisabeth A. Strunk     John C. Knight

*Department of Computer Science*

*University of Virginia*

*151 Engineer's Way, Charlottesville, VA 22904-4740*

*{strunk | knight}@cs.virginia.edu*

## Abstract

*It is often the case that safety-critical systems have to be reconfigured during operation because of issues such as changes in the system's operating environment or the failure of software or hardware components. Operational systems exist that are capable of reconfiguration, but previous research and the techniques employed in operational systems for the most part either have not addressed the issue of assurance or have been developed in an ad hoc manner. In this paper we present a comprehensive approach to assured reconfiguration, providing a framework for formal verification that allows the developer of a reconfigurable system to use a set of application-level properties to show general reconfiguration properties. The properties and design are illustrated through an example from NASA's Runway Incursion Prevention System.*

## 1. Introduction

In order to ensure system safety, mission success, or other crucial system properties, safety-critical systems often must be reconfigured during operation. This reconfiguration might be required as a response to many stimuli such as changes in the system's operating environment or the failure of software or hardware components. Examples include failure of an aircraft fly-by-wire system, damage to control surfaces [2], and the loss of a processor in an integrated modular avionics system.

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 [16]. For the most part, however, previous research and the techniques employed in operational systems either have not addressed the issue of assurance of critical reconfiguration properties or have been developed in an ad hoc manner to meet the needs of specific systems. In safety-critical systems, it is essential that the actions involved in reconfiguration be completed correctly, on time, and with

very high levels of assurance. A general approach that can be instantiated for specific applications is highly desirable.

In this paper we present a comprehensive approach to *assured* reconfiguration. Our goal is to provide a set of design level properties which, when shown of a system specification, together imply that the system's reconfiguration will take place with high dependability. We start by introducing a formal definition of reconfiguration and an associated set of high-level, general properties. We then outline an architecture that will support assurance of those properties, provide a set of design-level properties that can be shown of that architecture, and show rigorously that the design-level properties imply the general reconfiguration properties. Showing for a specific system that the design-level properties have been met will then imply assurance of reconfiguration for that system.

Having shown that the design-level properties imply the high-level reconfiguration properties, it then remains to indicate how a system developer might go about proving design-level properties for a specific system. To do this we use an example from the avionics domain: NASA's Runway Incursion Prevention System (RIPS), an experimental system designed to detect runway incursions in airports. We choose certain elements of the RIPS specification and explain how their associated reconfiguration design properties can be demonstrated.

Section 2 elaborates the motivation for this work, and Section 3 defines its applicability. Section 4 gives the general properties of a reconfiguration process, and Section 5 presents a candidate architecture through which these properties might be assured. Section 6 details this architecture, and Section 7 outlines a set of proofs that the architecture has the properties set out in Section 3. Section 8 gives an example implementation, Section 9 suggests tool support for aiding this process, and Section 10 concludes the paper.

## 2. Motivation for Reconfiguration Assurance

Complex software systems frequently operate in heterogeneous environments with complex goals including high

dependability requirements. These goals are often independent (and sometimes even conflicting), and building a single software system that will achieve them can be very difficult. Furthermore, goals such as overall system safety can be difficult to meet, and showing that they have been met is still more demanding. The state of the art in software development has stood the engineering community in good stead for the most part, but there have been notable failures (e.g., the Ariane V [7]), and complexity is steadily increasing.

A current research direction that is addressing this problem is to build complex systems that can operate in more than one logical configuration where those different configurations do not necessarily provide the same functionality. The concept is to transition to different configurations in response to detected failures in system components that could compromise the system's dependability goals, rather than trying to mask the effects of the component failure by some form of replication. In systems where safety is a major goal, such as the survivable systems proposed by Strunk and Knight [14], those using the graceful degradation framework of Shelton, Koopman, and Nace [12], or those using the Simplex architecture of Sha [10], this approach allows the system design to be much simpler and therefore easier to analyze. Reconfiguration to assist dependability is also carried out in current systems such as the Boeing 777 [16] and the Lockheed-Martin F-22 [9].

Reconfiguration can also aid the construction of software systems where the system must be dependable but reconfiguration does not support this explicitly. Such systems appear frequently in space missions (e.g., the Corot mission [3]), in "intelligent" control systems (such as described by Stewart et al. [13] and Bateman et al. [2]), and in the more general case of adaptive reconfigurable computing [8, 6]. In these kinds of applications, the software must carry with it an assurance argument; part of that assurance argument must be that the reconfiguration aspects will function properly.

Much of the existing literature on reconfiguration either discusses reconfiguration with only a superficial treatment of assurance, or focuses only on hardware platforms and how to distribute software processes over available elements. We focus explicitly on reconfigurable software, showing how to assure properties of an application that has a set number of discrete, predefined functional sets.

### 3. System Assumptions

Before attempting to assure reconfiguration, we must define precisely what is to be assured. We assume a class of systems where the functionality that provides value to the user can be identified before system deployment and is set forth in a set  $S$  of specifications of different levels of function. Reconfiguration is not defined over arbitrary system configurations, but rather between two specific members of

$S$  (denoted  $S_i$  and  $S_j$ ). This is appropriate in the context in which one must assure reconfiguration correctness, because assuring reconfiguration correctness is secondary to assuring at least some basic level of specification correctness.

We also assume that worst-case execution times, including worst-case time to train data in control systems, can be determined for each function in a specification. This is a standard requirement of highly dependable systems.

Third, we assume that the system in question consists of independent processes. Process interaction during reconfiguration depends on individual process reconfiguration; we address the latter here and leave the former to future work.

Finally, we assume that the system can halt normal execution for the time that it takes to reconfigure. In some circumstances, the transition might take sufficiently long that specific action must be taken during the transition. Such action is not included explicitly in the definition, but can be addressed by assuming an intermediate specification,  $S_k$ , between  $S_i$  and  $S_j$ .  $S_k$  can encompass only minimal function so that any transition to it can have a very strict time bound. If the transition from  $S_k$  to  $S_j$  cannot be made in the required time, further intermediate specifications can be added.

### 4. Definition of Assured Reconfiguration

Reconfiguration in the sense in which we use it is, informally, *the process through which a system halts operation under its current source specification  $S_i$  and begins operation under a different target specification  $S_j$* . We address in this paper the requirements on a single reconfiguration; because the assurance arguments make no assumptions on previous reconfigurations, system operation can consist of an arbitrary sequence of these reconfigurations.

To formalize this definition, we must define a set of supporting elements. In this set, let  $S_x, S_y \in S$  (defined below),  $S_x \neq S_y$ . Functional and state properties are expressed in set theory, where appropriate; timing properties are expressed in Real-Time Logic (RTL) [5] (see Figure 1 for a brief summary of the features used here). To make our discussion clearer, we let  $a$  = the occurrence number of  $S_i$ ,  $b$  = the occurrence number of the reconfiguration, and  $c$  = the occurrence number of  $S_j$  in the single reconfiguration sequence we consider. The necessary supporting terms are as follows:

- $S$ : the set  $\{S_1, S_2, \dots, S_n\}$  of service specifications of the system
- $E$ : a set of possible functions from factors that affect which specification is an appropriate instantiation of  $S_j$  to possible values of those factors
- $Env(t) \rightarrow e \in E$ : function that returns the value of the environmental state at time  $t$
- $Choose(S_x, e \in E) \rightarrow S_y$ : function that returns appropriate

<b>Event</b>	A temporal marker, i.e., the occurrence of an event marks a point in time
<b>Action</b>	Schedulable unit of work
$\downarrow A$	The event marking the completion of action <b>A</b>
$\uparrow A$	The event marking the initiation of action <b>A</b>
$@(E, i)$	Occurrence function. $@(E, i)$ , $i \in \mathbb{N}_+$ , represents the time of the $i$ th occurrence of event <b>E</b> .
$Q \langle x, y \rangle$	Predicate <b>Q</b> is true before or at time <b>x</b> , and does not become false before time <b>y</b> [For clarity we have introduced a slight change in syntax]

**Figure 1. A Brief Overview of RTL (excerpted from Jahanian and Mok [5])**

- target elements of **S**, given source elements of **S** and **E**.
- $Pre_x$ : the precondition that must be satisfied for the system to operate under  $S_x$
  - $Inv_{ij}$ : an invariant that must hold during the transition from  $S_i$  to  $S_j$
  - $\uparrow S_x$ : the event marking the beginning of the system's operation under specification  $S_x$ . This event occurs when the system first operates according to the function set out in  $S_x$ , and concurrently satisfies  $Pre_x$ .
  - $\downarrow S_x$ : the event marking the end of the system's operation under specification  $S_x$  (we define the occurrence of a reconfiguration signal to imply  $\downarrow S_x$ )
  - $T_{ij}$ : the maximum allowable time between  $\downarrow S_i$  and  $\uparrow S_j$

More formally, then, reconfiguration is defined as the action **R** in which the following conditions hold:

- P1.**  $@(\uparrow R, b) = @(\downarrow S_i, a)$   
*(R begins at the same time the system is no longer operating under  $S_i$ )*  
 Note that this property defines the beginning of **R** rather than specifying a requirement that should be met.
- P2.**  $@(\downarrow R, b) = @(\uparrow S_j, c)$   
*(R ends at the same time the system becomes compliant with  $S_j$ )*
- P3.**  $\exists t$ : time s.t.  $@(\uparrow R, b) \leq t \leq @(\downarrow R, b) \wedge$   
 $(Choose(S_i, Env(t)) = S_j) \langle t, t \rangle$   
*( $S_j$  is the proper choice for the target specification at some point during **R**)*
- P4.**  $@(\downarrow R, b) - @(\uparrow R, b) \leq T_{ij}$   
*(R takes less than or equal to  $T_{ij}$  time units)*
- P5.**  $Inv_{ij} \langle @(\uparrow R, b), @(\downarrow R, b) \rangle$   
*(The transition invariant holds during **R**)*
- P6.**  $Pre_j \langle @(\downarrow R, b), @(\downarrow R, b) \rangle$   
*(The precondition for  $S_j$  is true at the time **R** ends)*
- P7.**  $@(\downarrow S_j, c-1) \langle @(\uparrow R, b) \rangle \Rightarrow @(\uparrow S_j, c) = @(\downarrow R, b)$   
*(The lifetime of **R** is bounded by any two occurrences of the same specification)*

## 5. An Architecture for Assurance

### 5.1 Reconfiguration Sequencing and Timing

The definition of reconfiguration given above is expressed as a set of general properties; it is applicable to a number of architectures, but too abstract for us to show how one might assure reconfiguration correctness for a specific system. In this section, we present 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 implementer builds a system using this architecture and shows the low-level properties required of the architectural elements, he will know that the high-level properties that assure reconfiguration have been met.

Our proposed architecture has two major components: (1) the *application* that performs the computations associated with the members of **S**; and (2) a *reconfiguration mechanism* that ensures reconfiguration can be carried out. In our logical model, the application is a process that runs continuously from the time the system begins operation to the time the system is shut down. This process can be in either functional modes defined by members of **S** or modes specific to reconfiguration. We assume that the application components that implement the elements of **S** possess the fail-stop property [11]. This might be ensured using safe programming [1] or some form of protection shell [15].

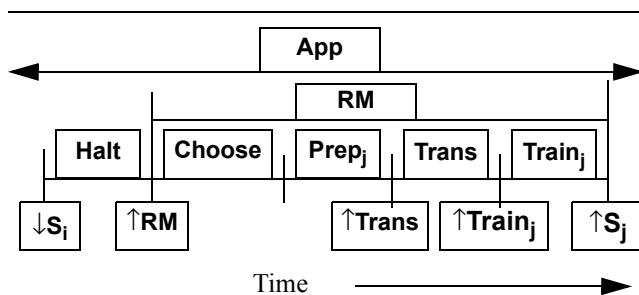
We give overall control of the reconfiguration process to the reconfiguration mechanism. This has two major benefits. First, while the reconfiguration mechanism will control the application, we can design it so that it is largely independent of the application; this greatly increases its potential for reuse. If it can be reused, the cost associated with formal verification, for instance, can be amortized over a number of systems. Second, much of the application-dependent complexity can be located within the application itself, leaving the reconfiguration mechanism as a fairly straightforward structure that can be reasoned about with a high degree of confidence. Deferring complexity to the application does not eliminate the complexity, but it does allow it to be encapsulated within the design components of the application (Sections 5.2 and 6 present a method for doing this). Also, in a system that tolerates faults through reconfigura-

tion, the reconfiguration mechanism must be more dependable than the application itself.

If we are to refine the properties given in Section 4, we first need building blocks for the statement of the refined properties. We define the following more specific events, actions, and predicates (also shown in Figure 2):

- App** Action representing operation of the application. Operates continuously throughout  $R$ .
- RM** Action representing operation of the reconfiguration mechanism.
- $\uparrow\text{RM}$  Event that **App** signals fault at top level
- $\downarrow\text{RM}$   $\equiv \downarrow\text{Trans}$  (below)
- $\downarrow\text{S}_i$  Event that a reconfiguration signal is sent to  $S_i$
- Halt** Action of **App** causing  $S_i$  to meet **Post**
- $\uparrow\text{Halt}$   $\equiv \downarrow\text{S}_i$
- Post** Predicate that must be true of **App** in order for reconfiguration to begin. This condition protects data, and ensures that the software is fail-stop.
- Choose** Action of **RM** determining member of  $S$  for  $S_j$
- Pre<sub>transj</sub>** Predicate that must be true of **App** before passing control back to **App**
- Prep<sub>j</sub>** Action of **App** causing **Pre<sub>transj</sub>** to be met
- Trans** Action of **App** effecting functional transition
- $\uparrow\text{Trans}$  Event that **RM** instructs **App** to transition to  $S_j$
- $\downarrow\text{Trans}$  Event that **App** acknowledges to **RM** that transition is complete
- Train<sub>j</sub>** Action of **App** initializing  $S_j$  or training its data
- $\uparrow\text{Train}_j$   $\equiv \downarrow\text{Trans}$
- $\downarrow\text{Train}_j$   $\equiv \uparrow\text{S}_j$
- $\uparrow\text{S}_j$   $S_j$  has now initialized or trained all data and is operational.

We assume for clarity that all actions associated with  $S_i$  have its occurrence number (a); all actions associated with  $S_j$  have its occurrence number (b); and all other actions have the occurrence number of  $R$  (c). This is a formalization of the informal idea that the actions are those that occur during a particular reconfiguration.



**Figure 2. Actions and Events in Application and Reconfiguration Mechanism**

Our temporal structure assumes that additional reconfiguration signals are not generated during the reconfiguration process. This is reasonable since no reconfiguration signals can be generated by the application during reconfiguration, and if an external reconfiguration signal is generated during reconfiguration, that signal can be buffered until control is passed back to the application during the training period for  $S_j$ . If time does not permit this, the reconfiguration mechanism must have a signal handler that will restart it during reconfiguration. Because the application postcondition would already be met, assuring reconfiguration from the reconfiguration mechanism is a simpler case than assuring reconfiguration from the application; thus, we do not address its details here.

## 5.2 Detailed Application Structure

The structure we use for the application to enable its reconfiguration assurance properties to be met is motivated by the temporal model presented above and the need to assure the properties presented in Section 2. We define properties that must be provided by the application in this section so that we can refer to them when discussing the reconfiguration mechanism and when showing that the application and the reconfiguration mechanism together meet the general reconfiguration properties.

The application is built of a set of modules, each of which has an interface designed to support reconfiguration assurance. The modules are design components that follow decomposition rules such as would be used in building an implementation of a nonreconfigurable system. Properties of the application are decomposed into properties of the modules, so that conjoining module properties (e.g., postconditions) gives the corresponding application property.

Each interface function will take a module-specific *service level parameter*. The service level parameter instructs the interface to provide a specific level of function, ranging from basic safe service to more elaborate calculations or operations for full function. It is the composition of different module service levels that allows the system to operate under different specifications.

Several options exist for effecting module reconfiguration. We choose to make module service level independent of the operational specification and to pass the service level parameter as part of an interface call. This strategy allows modules to be constructed and analyzed independently, and gives more flexibility in that different service levels can be used within the same operational specification, where appropriate. Also, the designer of the calling module in this case will know exactly which functional capability he is calling because the request for a specific functional capability will be passed to the function directly. Finally, the reconfiguration time for the functional aspects of the system

vanishes since the functional configuration is established dynamically at the time of a call.

Each module's interface might include checks on safety properties whose violation will trigger reconfiguration (similar to the structure of a safety kernel [15]) or some other reconfiguration triggering mechanism, but this is not necessary. We model our requirements for the case where signals can be generated internally since an external interrupt can be transferred through this mechanism as well.

## 6. Architecture Capabilities and Properties

The full formal characterization of all the design-level properties that must be shown of a specification in order to be sure it meets the required high-level reconfiguration properties is both complex and lengthy, as well as somewhat specific to the formalism used. Here, we state rigorously but informally the necessary design-level properties that must be shown, including their relationship to the general RTL elements defined above where appropriate. We use this mix of informal and formal characteristics to argue rigorously that these properties imply those from Section 4. A system developer can then characterize these properties in his particular formalism in order to show them for the formal expression of the application's function.

All of the following capabilities and properties are predicated on a reconfiguration trigger, i.e.,  $\exists t$ : time such that  $t = @(\downarrow S_i, a)$ . It is often simple to show a property without the condition but sometimes impossible since many events will not happen without the trigger. We leave this condition as implicit for clarity, but take advantage of it when needed.

### 6.1 Module Capabilities

Each module interface must provide certain capabilities:

- *An interface to the set of functions contained within the module.* A module can be accessed only through the functions in its interface. Those functions, however, can represent multiple service levels.
- *A set of possible values for the service level parameter.* These values define the service sets for the module.
- *A set of persistent data structures.* This is the data which is relevant to preconditions, postconditions, and invariants. If it is empty, preconditions, postconditions, and invariants are defined to be *true*.
- *A module postcondition.* This is a basic coherency condition representing the minimum state requirement for the application to continue some form of operation.
- *A mechanism through which reconfiguration signals are handled.* This mechanism will effect internal state restoration since the state might be inconsistent when a reconfiguration is signaled. It will leave the state consistent

with the module postcondition through methods such as forward or backward error recovery.

- *A mechanism through which reconfiguration signals are propagated to calling functions.* Each interface function has as its range the union of the natural range of the function together with a signal value representing a reconfiguration request.
- *A set of module transition conditions.* Each member of this set is a predicate that must be met before the module can begin training of the data associated with a member of  $S$ .
- *A set of module preconditions.* Each member of this set is a predicate that must be met in order for the module to have begun operation under an associated member of  $S$ .
- *A mechanism through which a module's service transition condition is guaranteed to be met.* If the transition condition of a module for its use under  $S_j$  is not the same as the postcondition for the module, a separate *prep function* to be called after  $S_j$  has been determined must be provided. If the transition condition can vary across members of  $S$ , the function must take a parameter specifying the appropriate version of the function.
- *Timing guarantees on interface functions related to reconfiguration.* These timing guarantees will be used to show overall reconfiguration timing guarantees. For the purely functional aspects of the system they will not pose an additional burden since they would be required to calculate the time requirements of an operation regardless of whether the system were reconfigurable.
- *A set of assured reconfiguration invariants.* These show little difference from standard application invariants except that they are likely to affect solely the module data that is externally visible. The set of invariants, in order of strictness, includes: (1) those on operation within each member of  $S$ , which must be satisfied until the module is sent a reconfiguration order to satisfy its postcondition; (2) those on transitions out of each member of  $S$ , which must be satisfied until the module is sent a reconfiguration order specifying  $S_j$  and ordering the module to meet its precondition; and (3) those on transition between any two possible successive members of  $S$ , which must be enforced from the time the module is instructed to meet its precondition to the time the next functional demand from the application arrives.

The properties of the modules that must be shown using these capabilities are set out in Table 1.

### 6.2 Application Capabilities

The modules that compose the system are contained within a separate top-level structure that we refer to as the *monitoring layer*. The monitoring layer includes:

<i>M1</i>	If none of a module's functions is currently executing, that module's postcondition is met.
<i>M2</i>	Each module has a function <code>halt</code> that, when called, will cause its postcondition to be met.
<i>M3</i>	Each module function either: always leaves its data in a consistent state; or when interrupted, calls the module <code>halt</code> function to leave the state consistent with the module postcondition.
<i>M4</i>	If a function is interrupted, its caller is interrupted with no intervening calls to any function other than <code>halt</code> .
<i>M5</i>	There is a method to meet the transition condition for each specification level.
<i>M6</i>	Each function always meets its timing constraint.
<i>M7</i>	The invariant for normal operation is stricter than the generic reconfiguration invariant, which is stricter than the specific reconfiguration invariant: $Inv_i \Rightarrow Inv_{ix} \Rightarrow Inv_{ij}$

**Table 1: Module Properties**

- *A facility to activate the reconfiguration mechanism.* When a reconfiguration signal has been propagated to the monitoring layer, the monitoring layer must be able to cause the reconfiguration mechanism to begin operation.
- *A state variable `config` representing the current operational specification.* This is the only place information on the exact member of `S` is stored. It contains the result of the calculation in `Choose()`.
- *The capability to cause operation under the current specification.* The monitoring layer is capable of initiating subactions of `App` so that `App` can take advantage of configuration to call appropriate module service levels.
- *Maximum time `train_time` that training of data might take for each member of `S`.* This will be clearly an estimate, but because training is considered to be part of the transition, it is necessary to state the worst case time.

The properties of the application that must be shown using these capabilities are outlined in Table 2.

### 6.3 Reconfiguration Mechanism Capabilities

Now that we have established the structure of the application, we are able to create a structure for the reconfiguration

mechanism that will use the application properties to meet the conditions set out in Section 4. Required elements of the reconfiguration mechanism are:

- *An implementation of `Choose()`.* The result will be stored in `config`.
- *A mechanism through which each module is ordered to meet its precondition for the new service level.* This mechanism will call functions in all individual modules with preconditions on internal state, ordering them to execute the version of that function appropriate to the current service level.

The properties of the reconfiguration mechanism that must be shown using these capabilities are outlined in Table 3.

## 7. Reconfiguration Assurance Argument

The properties of events and actions that will occur in a reconfiguration in our architecture are a refinement of high-level reconfiguration properties laid out in Section 4. In this section, we use the design-level properties stated in Section 6 to show that, in a system using our architecture and satisfying the architectural properties, the high-level properties hold.

<i>App1</i>	If <code>App</code> is not reconfiguring, it will function in accordance with the specification represented by <code>config</code> 's value.
<i>App2</i>	Every operation is called from some sequence of functions initiated by the monitoring layer.
<i>App3</i>	The postcondition is the conjunction of module postconditions.
<i>App4</i>	The transition condition is the conjunction of module transition conditions.
<i>App5</i>	An interrupt of the monitoring layer will cause an immediate transfer of control to <code>reconfig</code> , which is the functional equivalent of the action <code>RM</code> : $@(\downarrow\text{Halt}, a) = @(\uparrow\text{RM}, b)$
<i>App6</i>	<code>config</code> 's value is invariant outside of <code>RM</code> : $\text{config} = x < @(\downarrow\text{RM}, b), @(\downarrow\text{RM}, b) > \Rightarrow \text{config} = x < @(\downarrow\text{RM}, b), @(\uparrow\text{RM}, b+1) >$
<i>App7</i>	There are no circular dependencies among module reinitialization functions.
<i>App8</i>	If the transition precondition holds at the time the transition is completed, $\text{Pre}_j$ will be met within <code>train_time(j)</code> time units: $\text{Pre}_{\text{transj}} < @(\uparrow\text{Train}_j, b), @(\uparrow\text{Train}_j, b) > \Rightarrow @(\downarrow\text{Train}_j, b) \leq @(\uparrow\text{Train}_j, b) + \text{train\_time}(j)$
<i>App9</i>	The transition takes no real time: $@(\uparrow\text{Trans}, b) = @(\downarrow\text{Trans}, b)$ This property is true of the structure rather than a particular application, and can be stated as an axiom.

**Table 2: Application Properties**

**P1.**  $@(\uparrow R, b) = @(\downarrow S_i, a)$

(*R* begins at the same time the system is no longer operating under  $S_i$ )

P1 is definitional only, and does not impose specific requirements.

**P2.**  $@(\downarrow R, b) = @(\uparrow S_j, c)$

(*R* ends at the same time the system becomes compliant with  $S_j$ )

This property is definitional and so by itself requires no proof. It implies that the system must at some point transition to  $S_j$ , but in our model this property is subsumed by P6 since  $\text{Pre}_j$  can in general be satisfied only after a functional reconfiguration takes place.

**P3.**  $\exists t: \text{time s.t. } @(\uparrow R, b) \leq t \leq @(\downarrow R, b) \wedge$   
 $(\text{Choose}(S_i, \text{Env}(t)) = S_j) < t, t >$

( $S_j$  is the proper choice for the target specification at some point during *R*)

We present the full proof of P3 to give the reader an example of their construction, and then outline subsequent proofs so that the reader can construct the full chain of reasoning for himself.

For brevity, we write  $(\text{Choose}(S_i, \text{Env}(t)) = S_j) < t, t >$  as the predicate  $\text{valid}_j(t)$ . We first establish that  $\text{valid}_j(t)$  is true for the time  $\text{Choose}$  ends:  $\text{valid}_j(@(\downarrow \text{Choose}, b))$ . RM1 says that

$\text{choose} \equiv \text{Choose}$

$\Rightarrow$  (by the relationship we have assigned functional and sequence properties)

$\text{choose.post} < @(\downarrow \text{Choose}, b), @(\downarrow \text{Choose}, b) >$

$\Rightarrow$  (RM2)

$(\text{Choose}(S_i, \text{Env}(t)) = S(\text{config})) < @(\downarrow \text{Choose}, b), @(\downarrow \text{Choose}, b) >$

App6 and RM3 tell us that  $(\text{config} = j) < @(\downarrow \text{Choose}, b)$ ,  $@(\downarrow \text{Choose}, b) >$  since  $S_{\text{config}}$  will be the new operational specification and  $\text{config}$  will remain constant until  $\uparrow S_j$ . Together with the above statement we see

$\text{valid}_j(@(\downarrow \text{Choose}, b))$ .

Intuitively, it is obvious that  $@(\uparrow R, b) \leq @(\downarrow \text{Choose}, b) \leq @(\downarrow R, b)$ . More rigorously:

$\exists t: \text{time s.t. } @(\downarrow \text{Choose}, b) = t \wedge \text{valid}_j(t)$

$\Rightarrow$  (axiomatic basis of time in RTL)

$\exists t: \text{time s.t. } @(\uparrow \text{Choose}, b) \leq t \leq @(\downarrow \text{Choose}, b) \wedge \text{valid}_j(t)$

$\Leftrightarrow$  (RM1 and RM5)

$\exists t: \text{time s.t. } @(\uparrow \text{RM}, b) \leq t \leq @(\uparrow \text{Trans}, b) \wedge \text{valid}_j(t)$

$\Leftrightarrow$  (App5 and RM8)

$\exists t: \text{time s.t. } @(\downarrow \text{Halt}, a) \leq t \leq @(\uparrow \text{Train}_j, c) \wedge \text{valid}_j(t)$

$\Rightarrow$  (axiomatic basis of time in RTL)

$\exists t: \text{time s.t. } @(\uparrow \text{Halt}, a) \leq t \leq @(\downarrow \text{Train}_j, c) \wedge \text{valid}_j(t)$

$\Rightarrow$  (definitions of Halt and Train)

$\exists t: \text{time s.t. } @(\downarrow S_i, a) \leq t \leq @(\uparrow S_j, c) \wedge \text{valid}_j(t)$

$\Rightarrow$  (P1 and P2, substituting for  $\text{valid}_j(t)$ )

$\exists t: \text{time s.t. } @(\uparrow R, b) \leq t \leq @(\downarrow R, b) \wedge$   
 $(\text{Choose}(S_i, \text{Env}(t)) = S_j) < t, t >$

**P4.**  $@(\downarrow R, b) - @(\uparrow R, b) \leq T_{ij}$

(*R* takes less than or equal to  $T_{ij}$  time units)

This property can be shown using P1, P2, the definitions of Halt and Train, App5, RM8, and RM6.

**P5.**  $\text{Inv}_{ij} < @(\uparrow R, b), @(\downarrow R, b) >$

(The transition invariant holds during *R*)

<b>RM1</b>	$\text{choose}$ will be executed immediately when RM is called. $\text{choose}$ is equivalent to the action $\text{Choose}$ . This means that $@(\uparrow \text{RM}, b) = @(\uparrow \text{Choose}, b)$ .
<b>RM2</b>	If the postcondition of $\text{choose}$ is met, the new operational specification is the correct one and is stored in $\text{config}$ .
<b>RM3</b>	If $\text{config}$ 's value is invariant outside of RM, then $\text{config}$ 's value is invariant outside of $\text{Choose}$ : $\text{App6} \Rightarrow (\text{config} = x < @(\downarrow \text{Choose}, b), @(\downarrow \text{Choose}, b) > \Rightarrow \text{config} = x < @(\downarrow \text{Choose}, b), @(\uparrow \text{Choose}, b+1) >)$
<b>RM4</b>	$\text{App7} \Rightarrow$ RM calls all the prep functions of the modules in an order in which no dependencies are violated.
<b>RM5</b>	Exactly the prep functions are called between $\text{choose}$ and transition to the monitoring layer; this implies: $@(\downarrow \text{Choose}, b) = @(\uparrow \text{Prep}_j, c) \leq @(\downarrow \text{Prep}_j, c) = @(\uparrow \text{Trans}, b)$ and $(\text{Post} < @(\uparrow \text{Prep}_j, c), @(\uparrow \text{Prep}_j, c) > \wedge \text{RM4} \wedge \text{App4}) \Rightarrow \text{Pre}_{\text{trans}_j} < @(\downarrow \text{Prep}_j, c), @(\downarrow \text{Prep}_j, c) >$
<b>RM6</b>	If each function meets its timing constraint, then App can halt, RM can execute, and App can train within the allotted time: $\text{M6} \Rightarrow @(\downarrow \text{Halt}, a) - @(\uparrow \text{Halt}, a) + @(\downarrow \text{RM}, b) - @(\uparrow \text{RM}, b) + @(\downarrow \text{Train}_j, c) - @(\uparrow \text{Train}_j, c) \leq T_{ij}$
<b>RM7</b>	The invariants that must hold during transition hold at the appropriate times: $\text{Inv}_i < @(\uparrow \text{Halt}, a), @(\uparrow \text{Halt}, a) > \wedge \text{Inv}_{ix} < @(\uparrow \text{Halt}, a), @(\downarrow \text{Choose}, b) > \wedge \text{Inv}_{ij} < @(\downarrow \text{Choose}, b), @(\uparrow S_j, c) >$
<b>RM8</b>	RM begins before transition and ends at the time of transition; training begins at the time of transition: $@(\uparrow \text{RM}, b) \leq @(\uparrow \text{Trans}, b) = @(\downarrow \text{RM}, b) = @(\uparrow \text{Train}_j, b)$

**Table 3: Reconfiguration Mechanism Properties**

This property can be shown using P1, P2, M7, and RM7.

**P6.**  $\text{Pre}_j < @(\downarrow R, b), @(\downarrow R, b) >$

*(The precondition for  $S_j$  is true at the time  $R$  ends)*

The proof of P6 is more complex because it requires that a series of predicates be satisfied. The proof is aided by using the following lemmas:

**L6.1.** An interruption will cause the application to meet its postcondition:  $\exists t: \text{time s.t. } @(\downarrow S_i, a) = t \Rightarrow \exists t: \text{time s.t. } \text{Post} < t, t >$

This lemma can be shown through induction using App2, App3, M1, M2, M3, and M4.

**L6.2.** An interruption will cause control to be transferred back to the monitoring layer:  $\exists t: \text{time s.t. } @(\downarrow S_i, a) = t \Rightarrow \exists t: \text{time s.t. } t = @(\downarrow \text{Halt}, a)$ .

This lemma can be shown through induction using App2 and M4.

Together with a second application of M1, these lemmas imply that an interruption will cause control to be transferred to the monitoring layer at the same time the postcondition is met. This being true,

**L6.3.**  $\text{Post} < @(\uparrow R, b), @(\uparrow \text{Pre}_j, c) >$

which follows from App5, RM1, RM5, and M1. Using RM5 again, then App9 and App8, we see that at some time  $t$  between  $@(\downarrow \text{Trans}, b)$  and  $@(\downarrow \text{Trans}, b) + \text{train\_time}$ ,  $\text{Pre}_j$  is satisfied. Because RM2 and RM3 tell us that  $\text{config}$  holds the correct value, and RM5 and App9 tell us that at  $\downarrow \text{Trans}$  control is passed back to the monitoring layer, and using App1 this means that the system is in functional compliance with  $S_j$ , we know that **App** is operating according to  $S_j$ , so  $t = \uparrow S_j$ ; and using P2, P6 is shown.

**P7.**  $@(\downarrow S_j, c-1) < @(\uparrow R, b) \Rightarrow @(\uparrow S_j, c) = @(\downarrow R, b)$

*(The lifetime of  $R$  is bounded by any two occurrences of the same specification)*

P7 is definitional only, and does not impose specific requirements.

Some of these properties can also be considered to be conditional. For instance, the timing constraints apply only when functional and data constraints have been met. Timing constraints, then, can be calculated with the assumption that preconditions and postconditions are true.

## 8. Example

### 8.1 The RIPS System and the RSM

As an example of how the module, application, and reconfiguration properties can be shown of a system, we look at NASA's Runway Incursion Prevention System

(RIPS). RIPS is an avionics application designed to run on an aircraft during takeoff or landing. It is intended to help pilots with situational awareness of runway traffic to prevent collisions with objects or aircraft on the runway. Specifically, we examine the Runway Safety Monitor (RSM) algorithm for RIPS [4], which takes input data on runway traffic and determines whether that data indicates the risk of a collision. Even the RSM component of RIPS is a complex system, however, and discussing all aspects of our approach to its design and assurance is beyond the scope of this paper. To illustrate the concepts behind the approach to assured reconfiguration that we have defined, we describe certain parts of the RSM and outline example fragments of the assurance argument.

While the RIPS system does not control the aircraft directly, it does give critical advice to pilots during takeoff and landing, the most accident-prone stages of flight. A failure can have a serious impact on safety if the system does not alert pilots to its failure; the pilots will expect to be notified of possible collisions, and might not be as alert as they would be did they know they were making safety decisions without the system's advice. The functions in the RSM involve some relatively complex geometric calculations, and also the data sent to the system is not always reliable, so it might be the case that the system cannot always operate as desired. We therefore have constructed our model of the system to have a set of service modes that provide varying levels of functionality.

We have built a model of the RSM algorithm that contains four major operational specifications. The first,  $S_1$ , monitors not only the runway itself, but also a defined area surrounding the runway when detecting potential collisions. The geometry of the structure of the surrounding area is relatively complex when compared to the simple rectangle of the runway. If the software can no longer achieve this function due to, for instance, loss of computing power or a software fault, then the application can reconfigure to  $S_2$ , which monitors only the runway proper. If for some reason even this cannot be achieved, then the system chooses one of two options: either it halts and merely alerts the pilot ( $S_3$ ) or it gives the aircraft a command to climb and alerts the pilot, readying the aircraft for the pilot to execute a missed approach if he needs time to adjust to the loss of situational information ( $S_4$ ).

### 8.2 Structure of the RSM

The RSM specification has three major modules (shown in Figure 3): GEOM, which computes basic geometric functions; IZ, which sets up the geometry specific to the RSM; and ALG, which analyzes incoming data with respect to those structures. GEOM is protected by a layer that checks its outputs to give them strong correctness arguments. There

are two sets of persistent state: the incursion zone structure (belonging to IZ), and the data structures for the system interface (belonging to ALG).

Any failure of a check will trigger a reconfiguration signal. As the signal is propagated through IZ, IZ causes the incursion zone structure to meet its postcondition. The same is true for ALG, of the system data structures. The monitoring layer takes care of periodic function initiation and also reconfiguration, when necessary. RM.choose calls the appropriate module functions to meet  $Pre_{transj}$ : it reinitializes the incursion zone structure.

The overall reconfiguration process is required to finish within one update data cycle, which in real time lasts 2000 ms. In order to facilitate a proof of the timing requirement, each function in the module interface and in the reconfiguration mechanism interface has an associated variable representing a timing bound. This variable will be instantiated after an implementation is created. Addressing timing this way gives an implementer freedom to partition the allowed time in a manner most useful to the implementation, while enabling detailed timing theorems of the specification to be created whose proof over an implementation will ensure the overall timing requirement is met.

### 8.3 Assuring Properties of the RSM

A full proof of every design-level property for the RSM specification cannot be shown here. In order to indicate how all of the high-level reconfiguration properties can be proved, we choose three representative design-level properties, one from each major structural element. We then explain how each can be shown for the RSM specification.

The representative module property we use is **MI**: If none of a module's functions is currently executing, that module's postcondition is met. While this property might seem difficult to assure overall, it can be simplified significantly by disallowing data structure access through any function outside the module interface; and that is statically checkable. In the RSM specification, only IZ can access the incursion zone structure and only ALG can access the

shared data structures. Each module interface function ensures that the postconditions are not violated.

The representative application property we use is **App1**: If **App** is not reconfiguring, it will function in accordance with the specification represented by config's value. The monitoring layer calls ALG, passing it a parameter representing the current operational specification. ALG then determines the service level of IZ that is appropriate for the specification, and for the function it is calculating. For instance, IZ has a function to check whether an input data target is in an incursion zone. If  $S_2$  is the operational specification, then ALG will always call this function with service level parameter  $a_2$ , asking it to check only the runway proper. If  $S_1$  is the operational specification, then ALG will call this function with service level parameter  $a_1$  or  $a_2$ , depending on various conditions of the monitoring aircraft and target. Each overall functional level  $S_i$  can be assessed independently by setting the specification parameter of ALG equal to  $i$ , and then determining what function sequences will be called for each function in ALG's interface.

The representative reconfiguration mechanism property we use is **RM7**:  $Inv_i < @(\uparrow Halt, a), @(\uparrow Halt, a) > \wedge Inv_{ix} < @(\uparrow Halt, a), @(\downarrow Choose, b) > \wedge Inv_{ij} < @(\downarrow Choose, b), @(\uparrow S_j, c) >$ . The reconfiguration invariants for the RSM are over the shared data structure, as this is the only way the RSM can affect the larger system. The transition invariant in general is that the reconfiguration process must not affect any of the elements of the shared data structure: it cannot begin or end a pilot alert, output possible collision targets, or instruct the aircraft to do anything other than remain on the course set by the operational specification. Such an invariant is a subset of the invariant of  $S_i$  because it is in fact determined by  $S_i$ , and it is the same for all possible target transitions  $S_j$ . It can be shown for the RSM by ensuring that no element of the reconfiguration mechanism accesses ALG's data structures, which is true because ALG's prep function does not access those structures.

The discussion above is an outline of a complete set of proofs that establish the high-level reconfiguration proper-

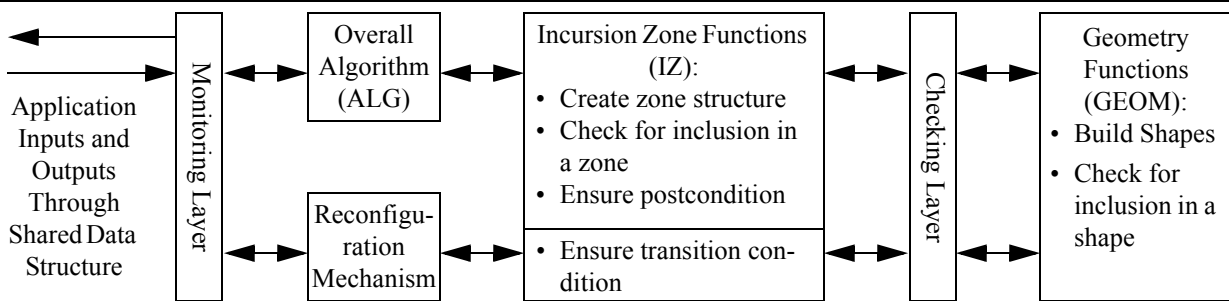


Figure 3. RSM Architecture

ties for the RSM part of RIPS. The level of assurance we can achieve by structuring the specification this way would be very difficult or impossible to reach using a standard approach using a single monolithic software design. Such a design would have to show that the entire system will function correctly, rather than that it can transition to a very simple set of functions that will achieve safety. Such a level of assurance would also be difficult to achieve if reconfiguration were employed, but with an ad hoc approach. All of the different circumstances under which a transition could occur would have to be analyzed separately. Our approach makes the assurance argument simpler, clearer, and more convincing.

## 9. Tool Support

Assurance arguments about complex software properties are not simple and cannot be made with the push of a button. Their construction and management can, however, be greatly simplified with appropriate tool support. Our framework facilitates this by defining the various architectural elements of a reconfigurable system and defining what those elements' properties should be. A variety of current theorem proving systems could be extended to allow developers to categorize the elements of a specification within this framework. For each of these elements, the system could then generate automatically (and in some cases, attempt to prove) the properties associated with that category of element.

## 10. Conclusion

The complexity of many current safety-critical applications and the strictures placed on them by their dependability requirements are increasing the prominence of reconfigurable system designs because of the opportunity those designs present to meet system dependability goals. Reconfiguration can in its own right introduce questions of correctness and assurance of correctness, however. We have presented a rigorous definition of reconfiguration assurance and an architectural framework through which an application might be constructed to meet the properties of that definition. This framework carries with it a set of design-level properties that, when shown of the application, mean the definition's properties have been met. We have listed these properties that are necessary to assure reconfiguration correctness and have outlined a strategy through which they might be met in practice.

## Acknowledgments

It is a pleasure to thank Phil Koopman for technical discussion on the subject of this paper. Martin Hiller also gave

us a number of helpful comments and suggestions. This work was sponsored, in part, by NASA under grant number NAG1-02103.

## References

- [1] Anderson, T., and R. W. Witty. "Safe programming." *BIT*, 18:1-8, 1978.
- [2] 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.
- [3] Cailliau, D., and R. Bellenger. "The Corot Instruments Software: Towards Intrinsically Reconfigurable Real-time Embedded Processing in Space-borne Instruments." Proc. 4th IEEE International Symposium on High-Assurance Systems Engineering, Nov. 1999.
- [4] Green, D.F. Jr. "Runway Safety Monitor Algorithm for Runway Incursion Detection and Alerting." NASA/CR-2002-211416, National Aeronautics and Space Administration, January 2002.
- [5] Jahanian, F., and A.K. Mok. "Safety Analysis of Timing Properties in Real-Time Systems." *IEEE Trans. on Software Engineering*, 12(9):890-904.
- [6] Karsai G., A. Ledeczki, J. Sztipanovits, G. Peceli, G. Simon, and T. Kovacszhazy. "An Approach to Self-Adaptive Software based on Supervisory Control." *Self-Adaptive Software*, Lecture Notes in Computer Science, 2002.
- [7] Lions, J. "Ariane V Flight 501 Failure: Report by the Inquiry Board." <http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf>.
- [8] Neema S., T. Bapty, and J. Scott. "Adaptive Computing and Run-time Reconfiguration." Proc. Military Applications of Programmable Logic Devices, Laurel, MD, September 1999.
- [9] Seeling, K. "Reconfiguration in an Integrated Avionics Design." 15th AIAA/IEEE Digital Avionics Systems Conference, Oct. 1996.
- [10] Sha, Lui. "Using Simplicity to Control Complexity." *IEEE Software* 18(4):20-28.
- [11] 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.
- [12] Shelton, C., P. Koopman, and W. Nace. "A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems." Eighth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003), Guadelajara, Mexico, Jan. 2003.
- [13] 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.
- [14] Strunk, E. A., and J. C. Knight. "Functionality/Dependability Co-design in Real-Time Embedded Software." Workshop on Co-design for Embedded Real-time Systems, July 2003.
- [15] Wika, K.G. and J.C. Knight. "On the Enforcement of Software Safety Policies." 10th Annual IEEE Conf. on Computer Assurance (COMPASS '95), June 1995.
- [16] Yeh, Y. C. "Triple-Triple Redundant 777 Primary Flight Computer." Proc. 1996 IEEE Aerospace Applications Conference, vol. 1, New York, N.Y., February 1996.