A Formal Model of Partitioning for Integrated Modular Avionics

Ben L. Di Vito

ViGYAN, Inc.
30 Research Drive
Hampton, VA 23666-1325

July 6, 1998

Prepared for NASA Langley Research Center

Contract NAS1-96014, Tasks DI07, DI16

DRAFT

Contents

List of Figures

Chapter 1

Introduction

The aviation industry is gradually moving toward the use of integrated modular avionics (IMA) for civilian transport aircraft. IMA offers economic advantages by hosting multiple avionics applications on a single hardware platform. An important concern for IMA is ensuring that applications are safely partitioned so they cannot interfere with one another, particularly when high levels of criticality are involved. Furthermore, IMA implementations would allow applications of different criticality to reside on the same platform, raising the need for strong assurances of partitioning.

NASA's Langley Research Center (LaRC) has been pursuing investigations into the avionics partitioning problem. This research is aimed at ensuring safe partitioning and logical noninterference among separate applications running on a shared Avionics Computer Resource (ACR). The investigations are strongly influenced by ongoing standardization efforts, in particular, the work of RTCA committee SC-182, and the recently completed ARINC 653 application executive (APEX) interface standard [1].

In support of this effort, we have developed a formal model of partitioning suitable for evaluating the design of an ACR. The model draws from the conceptual and mathematical modeling techniques developed for computer security applications. This report presents a formulation of partitioning requirements expressed first using conventional mathematical concepts and notation, then formalized using the language of PVS (Prototype Verification System). PVS is an environment for formal specification and verification developed at SRI International's Computer Science Laboratory [9]. A description of PVS is located on the World-Wide Web at URL `http://www.csl.sri.com/pvs/overview.html`. The system is freely available under license from SRI.

This work was performed in the context of a broader program of applied formal methods activity at LaRC [2]. Additional background and overview material on the use of formal methods in aerospace applications can be found in Rushby's formal methods handbooks [12, 13], and in a recent set of NASA guidebooks [7, 8].

Chapter 2

Avionics Computer Resource

The Avionics Computer Resource[1] (ACR) is an embedded generic computing platform that is able to host multiple applications (avionics functions), provide space (memory) and time (scheduling) protection for the applications as well as interrupt routing from a single source to the multiple applications. An ACR will be configurable and will apply to a wide range of aircraft types. The platform provides logical separation of applications present on the same ACR. It also provides a means to detect and annunciate any attempts to violate separation (fault containment).

2.1 Definitions

Several key terms are used throughout the following presentation. We collect them here to help clarify the conceptual model.

- **Applications:** Comprise the independent, active software entities (executable programs) that perform avionics functions.
- **Input ports:** Connection points from hardware devices external to the computing subsystem, e.g., sensors, cockpit switches and controls.
- **Output ports:** Connection points to hardware devices external to the computing subsystem, e.g., actuators, cockpit instruments and displays.
- **Resources:** Internal entities needed by applications to perform their functions, including processor execution time, memory space, disk space, communication links, etc.
- **Processors:** Hardware computing devices capable of executing or interpreting the software instructions issued by an application.
- **Kernel:** The core component of a processor's operating system, which is responsible for enforcing partitioning among applications in addition to other resource management functions.

[1] The term "resource" is overloaded in this domain. In the name "ACR," resource refers to a large structure composed of processor hardware and operating system software. Most of the time, however, we use the term resource to refer to smaller entities such as memory locations.

- **Services:** Individual functions or operations that may be requested from a kernel by either applications or higher layers of an operating system.

2.2 Operating System

A software operating system is a fundamental part of the ACR platform. Its purpose is to ensure that:

- The execution of an application does not interfere with the execution of any other application.
- Dedicated computer resources allocated to applications do not conflict or lead to memory, schedule, or interrupt clashes.
- Shared computer resources are allocated to applications in a way that maintains the integrity of the resources and the separation of the applications.
- Resources are allocated to each application independently of the presence or absence of other applications.
- Standardized interfaces to applications are provided.
- Software applications and the hardware resources needed to host them are independent.

The ACR operating system will provide highly robust, kernel-level services that may be used directly by the application developer or may serve as the basis for a more extensive operating system. In actual practice, the kernel services must be developed in accordance with the requirements of RTCA DO-178B [10] (or other applicable guidelines) and must be able to meet the highest level of criticality supported by DO-178B. While specific ACR implementations may be qualified to lower levels of integrity, kernel services must be sufficient to ensure isolation of applications of varying levels of criticality residing on the same ACR. The kernel services and the ACR itself must be qualified at or above the level of the most critical application allowed to reside on the ACR.

A key attribute of the ACR kernel is the requirement for a robust partition management mechanism. The partitioning mechanism underlies all aspects of the kernel. The kernel controls scheduling of partitions through a defined, deterministic scheduling regime (fixed round-robin algorithm or rate monotonic algorithm); controls communications between partitions; and provides consistent time management services, low-level I/O services, and ACR-level health management services. Figure 2.1 shows the ACR reference architecture assumed by SC-182.

To carry out its partitioning function, an ACR manages all hardware resources residing within the ACR and monitors access to all hardware resources connected to the ACR. The ACR kernel runs on the ACR hardware with sufficient control over all hardware and software resources to ensure partitions are noninterfering. The kernel performs an essential access mediation function that is independent of other services provided by the ACR. Access mediation must be complete, tamper-proof, and assured, where these attributes are defined as follows:

- **Complete.** There shall be no way for software running in any partition to bypass the kernel and access hardware resources not under the kernel's control.

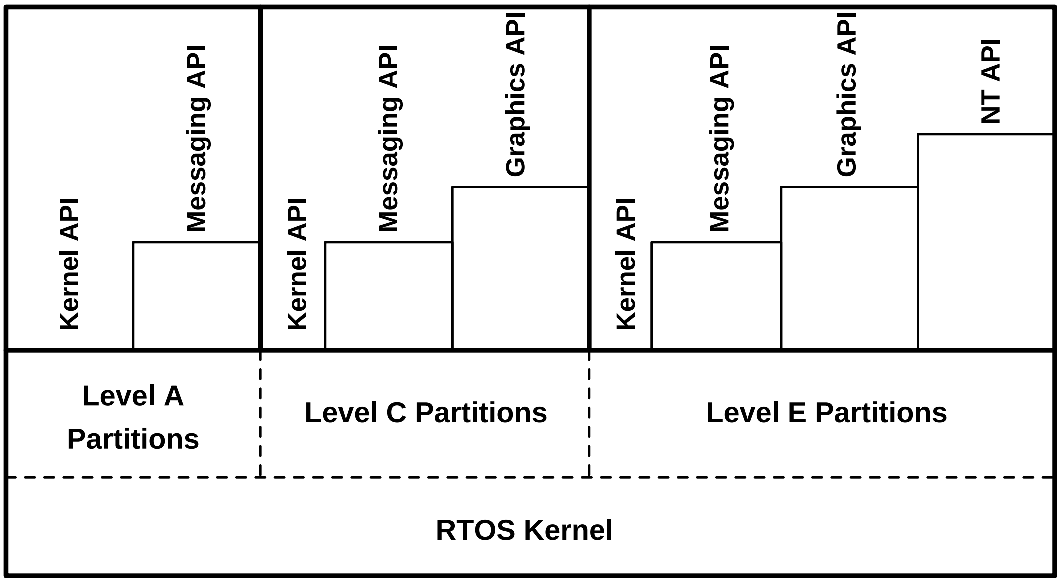

Figure 2.1: ACR Reference Architecture.

- **Tamper-proof.** There shall be no way for software in any partition to tamper with the kernel or its data so as to subvert the kernel's control of system resources.
- **Assured.** The kernel shall contain minimal functionality and shall meet all of the regulator's requirements for the criticality rating of the overall ACR.

Collectively these attributes ensure that an ACR has the minimum structural properties needed to achieve high-integrity partitioning. An ACR must possess these attributes regardless of which operating system services are offered to applications running within the ACR's partitions.

Some additional assumptions about the way applications are handled are listed below.

- A computing hardware platform is available that is capable of supporting basic operating system functions. The platform allows an executive to manage hardware and software resources; achieve separate execution domains and enforce separation of higher software layers; protect itself from faults, errors, or tampering that might defeat partitioning; and mediate all shared access to hardware and software resources according to an established policy.
- The kernel executes in its own protected domain with the highest privilege level available on the computer. Services are requested through a well-defined interface mechanism allowing users to pass parameters and receive results.
- Partitions define the boundaries of resource protection. If processes or tasks are provided within partitions, ACR resource protection is not extended to enforce separation among the processes or tasks.
- It is possible to have multiple instances of the same application within an ACR configuration. In such cases, each instance is considered separate and independent, and is protected from the other instances as if they were dissimilar applications.

Chapter 3

Formalizing Partitioning

We begin the formalization discussion by motivating the approach taken. A brief overview of the rationale is presented next. The complete formal models appear in Chapters 4 and 5. Note that the scope of the formal models is limited to issues of space partitioning. Time partitioning and other notions of separation are not covered in this report.

3.1 Security-Oriented Noninterference

Research in computer security has been active for nearly 30 years. Three broad problem areas are generally recognized by the security community: 1) confidentiality (secrecy), 2) integrity (no unauthorized modification), and 3) denial of service. Much study has been directed at DoD security needs, e.g., the "multilevel security" problem, which is primarily concerned with confidentiality. In this work, the motivation comes from an operating environment where multiple users access a common computer system. The users have different access permissions and the information they access is marked with different levels of sensitivity. The goal is to prevent users from viewing information they are not authorized to see.

While many security models have been devised to characterize and formalize security, researchers have had much success with the family of *noninterference models*. Originally introduced to address the confidentiality problem, these models can be applied to the integrity problem as well, which is the main concern in achieving space partitioning.

Noninterference models focus on the notion of programs executing on behalf of (differently) authorized users. Each such program affects the system state in various ways as instructions are executed. Users may view portions of the system state through these programs. What noninterference means in this context is that if user v is not authorized to view information generated by user u, then the instructions executed by u's program may not influence (or interfere with) the computations performed by v's program. In other words, no information that v is able to view should have been influenced by anything computed by u.

Goguen and Meseguer [4, 5] proposed the first noninterference model. Paraphrasing their model, the basic noninterference requirement can be stated as follows:

$$R(u, v) \supset O([[w]], v) = O([[P(w, u)]], v)$$

where $R(u, v)$ indicates that v may not view the outputs of u, $[[w]]$ is the system state that results after executing instruction sequence w, $P(w, u)$ is the sequence w with all of u's instructions purged from it, and $O(s, v)$ extracts from the system state those outputs viewable by v

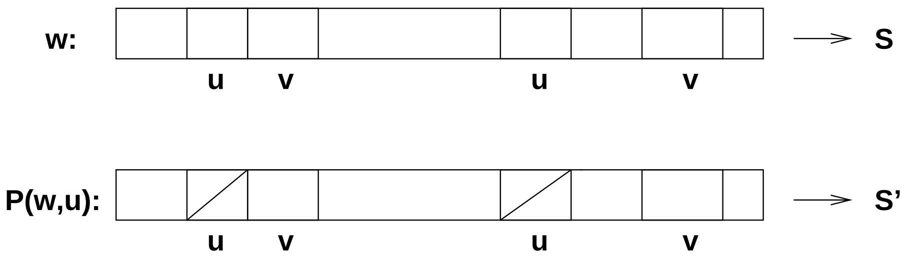

Figure 3.1: Instruction streams in the noninterference model.

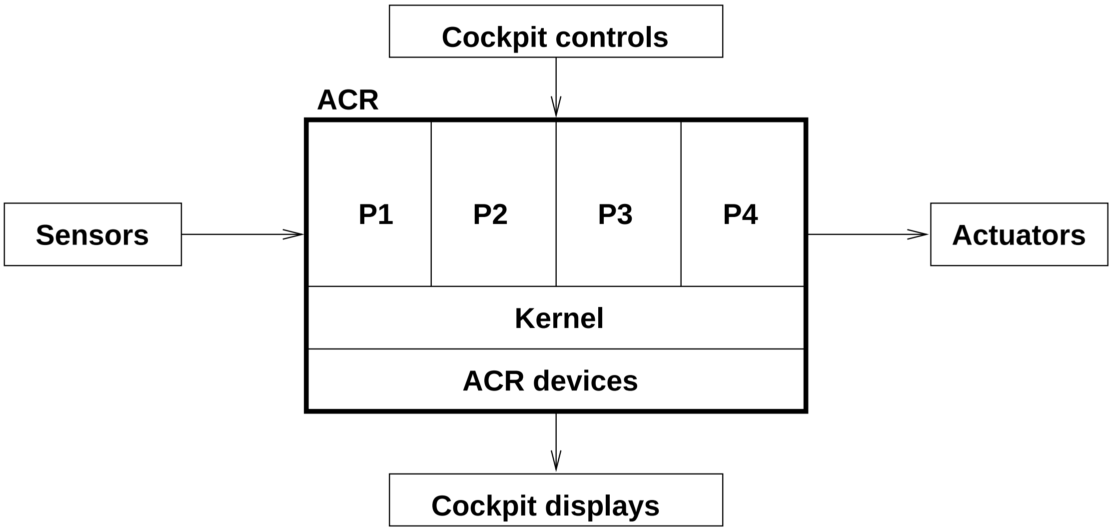

Figure 3.2: ACR devices (internal) vs. avionics devices (external).

(figure 3.1). What this assertion requires is that v's view of the state is the same regardless of whether u's instructions are executed. Hence, u cannot "interfere" with v. After Goguen and Meseguer's original formulation, other researchers introduced variations and extensions of their model for various purposes. Most important were the intransitive versions of noninterference formulated by Haigh and Young [6] and Rushby [11].

While the noninterference model is a powerful tool, its central requirement is too strong to be useful in a formalization of partitioning. The strict separation induced by this model is desirable in a security context, but is too confining in the IMA context. The reason is that communication between ACR partitions is expressly allowed, albeit under controlled conditions. Two types of communication can exist in an ACR environment: direct communication between partitions supported by operating system services, and indirect communication taking place through sharing or multiple access to avionics devices. For this reason, we make a distinction between "internal" (ACR) and "external" (avionics) devices (figure 3.2). The upshot is that it is permissible, under controlled conditions, for an application u to influence the computations of another application v, making a strict prohibition of "interference" too strong a requirement. It is possible to create a conditional noninterference model with suitable exemptions built in, but this runs the risk of exempting too much system behavior. Instead, the modeling approach we have selected draws from the essence of the noninterference concept and embeds it in a somewhat modified framework.

3.2 Conceptual Approaches

When considering the problem of formalizing partitioning, two broad approaches suggest themselves based on the nature of system modeling. One is a glass-box approach to modeling, where the outlines of internal system structure are known and explicitly represented. Conversely, a black-box approach abstracts away from the internal structure of the system and considers only the externally visible behavior of the system. We will find it useful to draw from both conceptual approaches in our adaptation of the noninterference concept.

3.2.1 Glass-box Approach

In the glass-box approach, we represent some key aspects of the internal structure of the ACR and the execution of applications. Typically, a state machine model is used to capture this structure and to formalize behavior within the system. The following steps are likely to be required:

- Model computer resources and applications.
- Model rules for resource allocation and access attempts by application software.
- Incorporate state machine to model the ACR.
- Specify partitioning as a property of instruction execution and OS service invocation—no improper accesses are allowed.

Using this approach, we would want to specify that applications access only those resources allocated to them and that the kernel maintains the separation of allocated resources.

3.2.2 Black-box Approach

In the black-box approach, we avoid representing aspects of the internal structure of the ACR and focus instead on the externally observable behavior that results from executing applications. Typically, an execution trace model is used to capture the observable behavior of the candidate system, where it is compared against the behavior of a reference system. The two alternatives are then required to be equivalent in some sense. In our case the two alternatives would be the desired integrated system and a fictitious federated system of equivalent functionality. A comparison of behavior between the two alternative systems is the means of capturing key system requirements. The following steps are likely to be required:

- Model ACR as a process with external port interfaces.
- Map ACR into an equivalent federated system (set of processes).
- Use a trace-based formalism to express process behaviors.
- Specify partitioning as a property that admits only behaviors that can be implemented on the federated system.

> **Fault tolerance:**
>
> **Distributed (replicated) system $\Longrightarrow$ Uniprocessor system**
>
> **Partitioning (noninterference):**
>
> **Uniprocessor system $\Longrightarrow$ Distributed (federated) system**

Figure 3.3: Relationship of RCP and ACR modeling approaches.

In essence, a partitioning property expressed in this manner asserts that no behaviors are allowed that take advantage of the multiple applications executing on a shared processor to compute results unobtainable in an equivalent federated system. This approach is somewhat indirect in that it does not actually prohibit the improper sharing of dedicated resources. What it does prohibit are computations that *depend* on such sharing, in effect, limiting the sharing to those cases where they have no impact on the outcome of the overall system computations.

3.3 Modeling Partitioning

The black-box approach is appealing because it allows for a more abstract, independent expression of partitioning requirements. This enables a broader variety of actual system designs to fit within the scope of the model. Unfortunately, by itself this modeling approach is inadequate to capture all that we need. Consequently, we will develop a model that has the flavor of the black-box concept at the higher levels while adopting the more direct glass-box approach to deal with representation of internal system structure.

Drawing on LaRC's work with the Reliable Computing Platform (RCP) [3], our modeling approach resembles the similar technique of comparison against a "gold standard" (figure 3.3). In RCP, a comparison between a distributed implementation and a single-processor implementation was used to formalize a notion of fault tolerance. In an analogous way, we use a comparison between a federated system and an integrated system to formalize a notion of noninterference for ACRs. Interestingly, the role of gold standard switched from the single processor system (in RCP) to the distributed system in the present model.

In both types of comparison:

- The application suites are the same.
- We are comparing the effects of running applications in two different execution environments.
- We are trying to rule out undesirable behaviors that might result when moving from the standard (assumed correct) architecture to the new (desired) architecture.

Before developing the model in full, we begin with a sketch of a formalization process that could be used to capture the high level concept of partitioning. The goal is to elucidate what we mean by partitioning and develop some intuition about the domain without yet introducing the full framework. The following steps summarize the key parts of the idealized method:

```
App1: loop                            App2: loop
        read sensor S into x;                 read sensor T into u;
        y := x + 1;                           v := u + 1;
        write y to actuator A;                write v to actuator B;
      end                                   end
```

Figure 3.4: Sample system containing two applications.

1. Given an ACR and its applications, map them into an equivalent federated system (each partitioned application in its own box).
2. Model the externally visible behavior of the ACR with execution trace T_0.
3. Assign traces $T_1, \ldots, T_n$ to the component behaviors in the federated system.
4. Require that if $L(T_1, \ldots, T_n)$ is the set of feasible interleavings of $T_1, \ldots, T_n$, then $T_0 \in L(T_1, \ldots, T_n)$ is a valid consequence.

What this scheme aims to do is rule out the presence of any observable behaviors in the ACR that cannot be duplicated, at least in principle, by an equivalent federated system. In other words, if the applications were migrated from a federated to an integrated architecture, no new system behaviors (modulo minor scheduling differences) could be introduced. One consequence of this approach is the limitation that certain memory sharing arrangements cannot be accommodated, e.g., many of those involving multiple readers and writers. Standard practice, however, in avionics architectures is to strictly avoid such sharing schemes, thus making the limitation moot in nearly all implementations.

3.4 Example Scenarios

To illustrate this high-level partitioning concept, we introduce a simple example of an ACR supporting two applications, `App1` and `App2`. Each carries out a cyclic computation of reading a value from a sensor, incrementing it, then sending the new value to an actuator. Figure 3.4 shows the structure of this sample system.

Let us assume for the sake of illustration that the ACR implementation has a flaw causing the memory space for some variables in the two applications to overlap. In particular, assume that both `y` from `App1` and `v` from `App2` are allocated the same space in memory. We will consider the effect of this flaw and how it might be manifested in the execution traces.

A typical schedule for running the two applications on an ACR would execute one loop iteration of `App1` followed by one iteration from `App2`. After each application has executed once, we would have a trace something like the following:

$$T_0 = \langle\, (S, 7), (A, 8), (T, 12), (B, 13)\, \rangle$$

These trace events record the observable inputs and outputs at the external avionics interface. Note that the assumed flaw has caused no incorrect behavior in this case.

If we now map these applications onto their federated system equivalent, each application would run on its own processor in the absence of the other. Rather than two interleaved instruction sequences, each machine would execute a single instruction stream from a single application. Assuming the same inputs are presented to the federated system interface, we would get the following traces for the two processors in the federated system:

$$T_1 = \langle\, (S,7), (A,8) \,\rangle$$

$$T_2 = \langle\, (T,12), (B,13) \,\rangle$$

Obviously, trace T_0 is a valid interleaving of the separate traces T_1 and T_2, reflecting our intuition that nothing improper occurred during this execution scenario.

Now consider a different way to schedule the execution of the two applications on the ACR. Let `App1` run only part way down its first iteration so that the assignment statement is executed but the write operation to the actuator is not. `App1` is suspended at this point and `App2` executes one complete iteration. Then `App1` is resumed and finishes its loop iteration. This schedule will result in the following trace:

$$T_0' = \langle\, (S,7), (T,12), (B,13), (A,13) \,\rangle$$

Notice how this schedule has exposed the implementation flaw. The value `App1` sends to actuator `A` is 13, not 8, due to the variable overlap condition. The value of 8 it had computed was overwritten by `App2`, and `App2`'s value is the one that was sent to the actuator. Notice further how it is impossible to obtain T_0' as a valid interleaving of T_1 and T_2. The actuator value of 13 from `App1` should not arise from a sensor input of 7. The computation represented by T_0' simply could not have occurred in the federated system.

Chapter 4

A Formal Model of Resource Partitioning

A formal model of ACR architectures and their partitioning requirements is presented below. The presentation begins with an informal discussion of the modeled entities and overall approach. Next, a generic model is introduced using informal mathematics. This is followed by three instances of the generic model expressed in the formal notation of PVS. The three instances investigate various system designs that include different combinations of features and their access control mechanisms. Excerpts from the PVS theories are presented below. The full text of the PVS theories can be found in Appendices A.1, B.1, and C.1.

We draw a distinction between partitioning requirements that apply to resources accessible to applications, and requirements that apply to private data held by the kernel. The overall partitioning model is divided into two parts based on this distinction. This chapter introduces the formalism for showing when application resources are protected from direct interference by other applications. Interpartition communication implemented by the kernel (or other ACR entities) presents the possibility of interference occurring within the kernel's domain. Chapter 5 develops the formalism for showing when the kernel can be considered free of flaws from this second type of interference.

4.1 Basic Framework

There are six aspects of modeling we will be concerned with: representation, computation, separation, requirement, policy, and verification. Each area is described below informally. Later this framework will be used to present the formalizations that follow.

- **Representation.** The basic entities of an ACR need to be represented, at least abstractly, as the first step in modeling. The set of applications or partitions is represented as a set of IDs or indices. Basic information units are assumed, corresponding to bits, bytes or whatever the smallest accessible unit happens to be. A *resource* space is assumed, which includes memory locations, elements of processor state, and memory-like devices. A *resource state* is considered to be a mapping from resources to information units. Finally, the basic computation step is denoted by the term *command*, which includes processor instructions, kernel service calls, and possibly other (atomic) operations.

- **Computation.** The elaboration of computation in the system is based on the *execution* of commands. Each command maps the values of some subset of resources into a set of results that are written to other resources. A sequence (list) of commands and the system's response constitute a computation. A recursive definition of system state resulting from the cumulative application of commands is constructed. Similarly, a computation trace containing the computed results from each command in a list is another view of computation.

- **Separation.** Given a notion of computation as the elaboration of a sequence of commands, the next step is to consider system responses to alternative command sequences. In particular, the command sequences formed by purging all commands but those belonging to a single partition will be considered. Taking the response traces resulting from the separate execution of all the purged command streams, we will compare them against segments of the integrated-system trace.

- **Requirement.** Having formed these trace pairs, one from the original command stream and the other from the purged command streams, we stipulate the partitioning requirement as equality of the two traces. If this condition is always obtained, then we know that the same set of computations will always result, whether performed in integrated or separated fashion.

- **Policy.** To achieve strong partitioning, it is necessary for the ACR to properly allocate resources and enforce access to those resources according to a suitable policy. The policy and system design are chosen to ensure that the partitioning requirement is always met.

- **Verification.** Finally, having modeled computation for the system features of interest, and captured the allocation and enforcement policy, it remains to show that the policy is a sufficient condition for the partitioning requirement. A proof is carried out to establish this result.

4.2 Mathematical Formulation

In this section we develop the key concepts of the partitioning model in the notation of informal mathematics. Later sections will present the formalized model cast in the PVS notation. The presentation below is only a sketch; many of the lower level details are deferred until the PVS formulation. What we wish to capture first is the essence of the model and its partitioning requirement for denoting the combined effect of separate execution.

4.2.1 Representation

Applications execute in an environment where the collective state of all applications running on an ACR is modified in response to each instruction or kernel service. This state includes main memory areas allocated to applications, state bits in the processor itself, and certain devices that have memory-like semantics. Individual elements of the state reside in a set of locations called resources, denoted R. The value held by a resource is an unspecified information unit drawn from the set I. The current resource state is given by a mapping $S : R \rightarrow I$.

Applications compute by executing commands, which include ordinary machine instructions (either native, emulated, or interpreted), kernel service primitives, and possibly other operations. Each is considered an atomic operation in the model, reading a set of arguments from the current state and writing a set of results to update the state. A command from the set K is a tuple (i, t, a, f, r), where i is the ID of the application invoking the command, t is the command type, a is a list of resources indicating the arguments to be read, f is a function with the signature $f : I^* \rightarrow I^*$ representing the computations on the arguments, and r is a list of result resources indicating where the result(s) of the function f should be written. Note that this representation of commands does not correspond to the way instructions are encoded and stored in memory, but is an abstract mathematical view of instructions. Lists of commands form the primary means of describing computations in the model.

As computations evolve, the results produced by each command and various other items can be imagined to form a *computation trace*. A trace event, drawn from the set E, contains the values computed by the command and some identifying information as well.

4.2.2 Computation

Execution of a command to produce a new value of the system state is modeled by the function $X : K \times S \rightarrow S$. We will forgo defining X until the PVS formulation. The current state is defined recursively by the cumulative application of X to a command list from K^* :

$$\begin{aligned} &S(\langle\,\rangle) = S_0 \\ &S(C \circ \langle\, k \,\rangle) = X(k, S(C)) \end{aligned}$$

where $\circ$ denotes the sequence or list append operation.

Construction of computation traces proceeds by applying the function $T : K \times S \rightarrow E$, which yields the computation event corresponding to a command's execution. The complete trace is defined recursively by the cumulative application of T to a command list from K^* :

$$\begin{aligned} &D(\langle\,\rangle) = \langle\,\rangle \\ &D(C \circ \langle\, k \,\rangle) = D(C) \circ \langle\, T(k, S(C)) \,\rangle \end{aligned}$$

Thus, we have for a command list C two key computational products: $S(C)$ is the state after executing all the commands in C, and $D(C)$ is the trace recording all the computed results. These values describe computation within the confines of a single processor, with instructions from different partitions interleaved in the list C.

4.2.3 Separation

Now we are ready to consider the mapping of the single processor (IMA) system into its equivalent federated system of multiple processors. Our goal is to take the same command stream and consider computation under two different architectures, integrated and federated. The method is to separate an integrated command stream into different threads of commands, one for each application (partition). Then computation is carried out separately for each individual thread, the results of which are compared against those generated by the integrated system.

First we provide a purge function to separate the original command stream into the different threads. $P : K^* \times A \rightarrow K^*$ denotes the purge function, mapping a command list C and application ID a into the appropriate subsequence of C. We overload the purge function by

$$
\begin{tikzcd}
C_i \arrow[r, "D"] & T_i \\
C_0 \arrow[u, "P"] \arrow[r, "D"'] & T_0 \arrow[u, "P"']
\end{tikzcd}
$$

Figure 4.1: Trace-based partitioning requirement.

adding a version of it for traces. $P : E^* \times A \rightarrow E^*$ extracts those elements of a computation trace belonging to application a. Again, we defer definitional details until the PVS form of the model.

4.2.4 Requirement

Having established the basic entities of the model, we are now ready to state the basic partitioning requirement. We focus on computation traces as the system response of interest. In the integrated system, the computation trace produced in response to a command list C is simply $D(C)$. We wish to compare portions of this trace to its analogs in the federated system.

When C is separated into subsequences based on partition, we have that the computation trace for case a is given by $D(P(C, a))$. Construct such a trace for each value a, then compare it to the subtrace found by purging the integrated trace $D(C)$. Thus the final partitioning requirement we seek has the form:

$$\forall a : P(D(C), a) = D(P(C, a))$$

The right hand side represents the computation applied to each command thread separately. Each processor in the federated system is assumed to be identical to the original, having the full complement of resources, although most will not be accessed (we hope) for a given choice of a.

Figure 4.1 illustrates the relationship of the various lists and traces in the manner of a classic commuting diagram, showing the familiar algebraic form of a homomorphism. In the figure we use C_0 to represent the original command list for the integrated system and $T_0 = D(C_0)$ its resulting computation trace. Then C_1 through C_n are the purged command lists and T_1 through T_n are their resulting traces.

If the access control policy of the system is working properly, then the effect of separation is invisible, yielding the same computation results as the integrated system. If, however, the policy or its enforcement is flawed in some way, one or more of the trace pairs above will differ, signaling a failure to achieve partitioning as formalized by this requirement.

4.2.5 Policy

With the help of protection features embedded in processor hardware, the kernel enforces an access control policy on the use of system resources. We denote by the predicate $H(C)$ the condition of command list C adhering to such a policy and other well-formedness criteria. The policy and type of enforcement are system dependent; it is not possible to be more explicit on the details without considering the design features themselves. The PVS models consider such details for three designs.

4.2.6 Verification

Pulling together all the pieces, we can now state the theorem needed to establish that an ACR design achieves strong partitioning:

$$H(C) \supset \forall a : P(D(C), a) = D(P(C, a))$$

A proof of this conjecture for all command lists C shows that the applications will be well partitioned under ACR control. This type of proof is constructed for all three PVS designs appearing in this report.

4.2.7 Variations

Variations on the basic outline sketched above are possible. We describe three variants here. Still others may suggest themselves based on particular needs.

The first variant concerns the form of the partitioning requirement. Sections 3.3 and 3.4 suggested the approach of taking the separate traces from the purged command streams and merging them back into a single trace. This "purge-and-merge" technique would lead to the following type of requirement:

$$D(C) = M(\lambda a : D(P(C, a)), N(D(C)))$$

where the function N extracts application IDs so the merge function M can reconstruct the trace in the correct order.

This approach is indeed workable and forms an intuitive statement of the partitioning condition. Nevertheless, the form we have chosen is simpler to work with and is equivalent to the "purge-and-merge" variant. We have constructed a proof in PVS to show this correspondence.

The next two variants concern the computational product used to compare the workings of the integrated and federated systems. We are using a computation trace for this purpose. One variant would be to use an external event trace instead. By this we mean a trace that records only events or data flows occurring at the ACR's external interface, what we have termed input and output ports. This type of trace was used in Section 3.4 and the preceding discussion. The computation trace we have described contains more information, including all the intermediate steps that would be omitted from an external event trace.

A requirement based on the external event trace would be a weaker constraint on system operation, but it would be sufficient to capture the essence of partitioning. It would allow designs that fail to agree on an instruction by instruction basis, but still matched on the important events of outputs to the avionics devices. By using the computation trace, we have a requirement that trivially implies the one based on external event traces. As a practical matter, it seems likely that the computation trace version will be applicable to nearly all designs.

The other variant concerns the use of system states instead of traces as the objects of comparison. The main requirement could be recast into an invariant showing that state values match in the two cases for suitably chosen resource and application ID combinations. These types of invariants must be proved anyway as part of the task of establishing the trace-based requirement.

The wisdom of this variant is not as clear as the other. Because the domain is aircraft control, the important matter is ensuring that outputs sent to actuators and other devices are correct. Traces are good for demonstrating this property. State invariants seem to fall one step short of what is needed. It is not enough to check that memory values match; what matters is what the system *does* with such values.

4.3 Illustration

To illustrate the partitioning model formulation, we return to the simple example of an ACR supporting two applications introduced in Section 3.4 (recall figure 3.4). As before, assume that the ACR implementation has a flaw. To be more specific, assume that variables `y` and `v` both have been allocated to memory location 1001. Now let us consider the effect of this flaw in terms of the formal model.

First, we need the command list to be executed by the IMA system.

$$\begin{aligned} C_0 = \langle\, & (A1, 0, 7, 1000), (A1, 1000, incr, 1001), (A1, 1001, id, 200), \\ & (A2, 0, 12, 1002), (A2, 1002, incr, 1001), (A2, 1001, id, 300) \,\rangle \end{aligned}$$

Here we have assumed that input ports `S` and `T` can be read like memory locations, and that two sampled values are fixed by the constant functions 7 and 12. Similarly, output ports `A` and `B` can be written like memory at locations 200 and 300. The identity function `id` is used in these commands to reflect data movement without modification.

Executing the command list C_0 and collecting the trace events using the model function D results in the following trace:

$$T_0 = \langle\, (A1, 7), (A1, 8), (A1, 8), (A2, 12), (A2, 13), (A2, 13) \,\rangle$$

This sequence shows the values computed by each instruction in the command list C_0.

Now, applying the purge function to C_0 produces the two purged command lists reflecting the separation of the two applications.

$$C_1 = \langle\, (A1, 0, 7, 1000), (A1, 1000, incr, 1001), (A1, 1001, id, 200) \,\rangle$$

$$C_2 = \langle\, (A2, 0, 12, 1002), (A2, 1002, incr, 1001), (A2, 1001, id, 300) \,\rangle$$

These, in turn, lead to the two traces below.

$$T_1 = \langle\, (A1, 7), (A1, 8), (A1, 8) \,\rangle$$

$$T_2 = \langle\, (A2, 12), (A2, 13), (A2, 13) \,\rangle$$

We find that $T_1 = P(T_0, A1)$ and $T_2 = P(T_0, A2)$, satisfying the partitioning requirement.

Next, consider the alternate command list below (C_0 reordered).

$$\begin{aligned} C_0' = \langle\, & (A1, 0, 7, 1000), (A1, 1000, incr, 1001), (A2, 0, 12, 1002), \\ & (A2, 1002, incr, 1001), (A2, 1001, id, 300), (A1, 1001, id, 200) \,\rangle \end{aligned}$$

The purged command lists will be the same as C_1 and C_2; hence T_1' and T_2' will be the same as T_1 and T_2. The integrated system traces, however, will be flawed due to the overlapping use of memory location 1001:

$$P(T_0', A1) = \langle\, (A1, 7), (A1, 8), (A1, 13) \,\rangle$$

$$P(T_0', A2) = \langle\, (A2, 12), (A2, 13), (A2, 13) \,\rangle$$

The condition $T_1' \neq P(T_0', A1)$ exposes the failure to achieve partitioning for command list C_0'.

4.4 ACR Design 1: Baseline System

This section introduces a PVS formalization of the basic partitioning model described in Section 4.2. The complete PVS theory can be found in Appendix A.1.

We assume a basic IMA architecture having the following characteristics:

- The system has a fixed set of applications.
- Only a single type of command exists: machine instructions.
- No interpartition communication is supported.
- Each resource is accessible by at most one application.
- Resource allocation and access rights are static (permanently assigned).

This configuration was chosen for the baseline because it represents a set of minimal system features. The next two designs will extend this baseline in important ways to show how more realistic designs can be accommodated.

4.4.1 Representation

Two uninterpreted types, `resource` and `info`, represent the space of resources (similar to addresses) and the basic information unit stored in resources. From these, the system state type is defined as follows.

```
resource_state: TYPE = [resource -> info]
initial_state: system_state
```

Resource state can be thought of as those portions of main memory allocated to applications plus a few other items depending on hardware design.

Commands are machine instructions, each having associated with it a list of argument resources (to be read), a function to be applied to the arguments, and a list of result resources (to be written).

```
cmd_fn:   TYPE = [info_list -> info_list]
cmd_type: TYPE+
command:  TYPE = [# appl_id: appl_id, cmd_type: cmd_type,
                    args: resource_list, fn: cmd_fn,
                    results: resource_list #]
cmd_list: TYPE = list[command]
```

Note that in applying the model to a real architecture, these commands may not correspond exactly to real instructions. Some features of actual processors, such as interruptible instructions and effective address calculations, may require the model to apply to imaginary microinstructions that would be composed to form the actual machine instructions. This is an interpretation detail, however, and does not affect the theoretical results obtained below.

Computation traces are constructed as lists of the `comp_event` type. Each event record contains the results computed by a single command.

```
comp_event: TYPE = [# appl_id: appl_id, cmd_type: cmd_type,
                      results: info_list #]
comp_trace: TYPE = list[comp_event]
```

Note that all traces are represented by lists using the predefined PVS list type. This works well but has one minor drawback. The normal method of list construction is to "cons" a new element on to the front (left) of an existing list. When used as traces, lists will appear in reverse chronological order when read from left to right. Nevertheless, this has no impact on our results because lists are used consistently throughout; it remains only a curiosity.

4.4.2 Computation

To represent the effects of command execution, we first need a function to update a list of resources within system memory with a list of values.

```
next_state(rlist: resource_list, values: info_list,
           s: system_state): RECURSIVE system_state =
   CASES rlist OF
     null: s,
     cons(r, rest): IF values = null
                      THEN next_state(rest, null, s)
                               WITH [(r) := null_info]
                      ELSE next_state(rest, cdr(values), s)
                               WITH [(r) := car(values)]
                    ENDIF
   ENDCASES
   MEASURE length(rlist)
```

Using the previous utility function, we can model command execution as follows, where the state is mapped over the argument resource list to retrieve values.

```
execute(c: command, s: system_state): system_state =
   next_state(results(c), fn(c)(map(s)(args(c))), s)

state(cmds: cmd_list): RECURSIVE system_state =
   CASES cmds OF
     null: initial_state,
     cons(c, rest): execute(c, state(rest))
   ENDCASES
   MEASURE length(cmds)
```

The current state is just the cumulative effect of applying `execute` recursively to the command list.

Turning to computation traces, we have two analogous functions for describing the result values after invoking a command, and then collecting all the events to form a trace.

```
do_step(c: command, s: system_state): info_list =
   fn(c)(map(s)(args(c)))
```

```
do_all(cmds: cmd_list): RECURSIVE comp_trace =
   CASES cmds OF
     null: null,
     cons(c, rest): cons((# appl_id := appl_id(c),
                             cmd_type := cmd_type(c),
                             results := do_step(c, state(rest)) #),
                          do_all(rest))
   ENDCASES
   MEASURE length(cmds)
```

4.4.3 Separation

The function `purge` is used to discard all commands from a list except those belonging to a single application.

```
purge(cmds: cmd_list, a: appl_id): RECURSIVE cmd_list =
   CASES cmds OF
     null: null,
     cons(c, rest): IF a = appl_id(c)
                       THEN cons(c, purge(rest, a))
                       ELSE purge(rest, a)
                    ENDIF
   ENDCASES
   MEASURE length(cmds)
```

This function could have been defined using the PVS primitive `filter`, but the explicit recursive definition simplified certain proofs. A second, overloaded version of `purge` is included to operate on computation traces.

4.4.4 Requirement

Now the basic partitioning requirement is easily expressed in PVS:

```
FORALL a: purge(do_all(cmds), a) = do_all(purge(cmds, a))
```

The quantifier `FORALL` will normally be omitted from this expression because PVS treats free variables in formulas as if they were universally quantified. For the special case of a single application, we have `purge(cmds,a) = cmds`, and the requirement reduces to `do_all(cmds) = do_all(cmds)`.

4.4.5 Policy

The access control policy for this design is straightforward. Read and write modes are independently supported. Each resource has an access control list (ACL) naming the applications that have access to it and in what mode(s). As before, this degree of granularity is different from what a kernel implementation would maintain, where a range of resources would likely be assigned to one ACL.

```
access_mode:  TYPE = {READ, WRITE}
access_right: TYPE = [# appl_id: appl_id, mode: access_mode #]
access_set:   TYPE = set[access_right]
```

This scheme works to describe uses of memory and some devices. Input devices could have read-only access while output devices would be write-only.

A predicate `alloc` declares the access control in effect for a given system. The following asserts key requirements about resource allocation, namely, that it be static (independent of state) and exclusive (only one application has access rights to a resource).

```
allocation: TYPE = [resource -> access_set]

static_exclusive(alloc_fn: allocation): bool =
   FORALL (r: resource):
      EXISTS (a: appl_id):
         FORALL (ar: access_right):
            member(ar, alloc_fn(r)) IMPLIES a = appl_id(ar)

alloc: {p: allocation | static_exclusive(p)}
```

Command lists adhering to this policy must satisfy the `proper_access` predicate below, which requires that for every command, the application has read access to all argument resources and write access to all result resources.

```
mode_access(m: access_mode, rlist: resource_list, a: appl_id): bool =
   FORALL (r: resource):
     member(r, rlist) IMPLIES
       member((# appl_id := a, mode := m #), alloc(r))

proper_access(cmds: cmd_list): RECURSIVE bool =
   CASES cmds OF
     null: true,
     cons(c, rest): mode_access(READ, args(c), appl_id(c))
               AND  mode_access(WRITE, results(c), appl_id(c))
               AND  proper_access(rest)
   ENDCASES
   MEASURE length(cmds)
```

4.4.6 Verification

Finally, we arrive at the point where we must prove that enforcement of the policy is a sufficient condition for the partitioning requirement.

```
well_partitioned: THEOREM
   proper_access(cmds) IMPLIES
      purge(do_all(cmds), a) = do_all(purge(cmds, a))
```

A completely mechanical proof of the theorem `well_partitioned` has been constructed using the PVS theorem prover. It relies on roughly ten supporting lemmas. A few typical lemmas are described below.

The lemma map_args asserts that if two resource states agree on all elements of a resource list, the effect of doing a lookup using map is the same.

```
map_args: LEMMA
     (FORALL r: member(r, rlist) IMPLIES s1(r) = s2(r))
   IMPLIES
     map(s1)(rlist) = map(s2)(rlist)

execute_in: LEMMA
     (FORALL r: member(r, args(c)) IMPLIES s1(r) = s2(r))
   IMPLIES
     (FORALL r: member(r, results(c)) IMPLIES
                  execute(c, s1)(r) = execute(c, s2)(r))
```

The lemma execute_in is central to the overall proof. It asserts that if two different system states agree on all resources used as arguments for a command, then the new states after executing the command will agree on all result resources. This lemma together with the access control constraints combine to imply that the following state_invariant always holds.

```
state_invariant: LEMMA
     proper_access(cmds) AND
     member((# appl_id := a, mode := READ #), alloc(r))
   IMPLIES
     state(cmds)(r) = state(purge(cmds, a))(r)

purge_step: LEMMA
   proper_access(cons(c, cmds)) AND a = appl_id(c) IMPLIES
     do_step(c, state(cmds)) = do_step(c, state(purge(cmds, a)))
```

This invariant plus the induction step lemma purge_step combine to prove the main theorem.

4.4.7 Alternatives

As described in Section 4.2.7, the partitioning requirement can also be expressed using a "purge-and-merge" style of formulation. The following definitions show how this can be accomplished.

Merging traces from a trace vector requires a list of application IDs to indicate the original command ordering. These IDs are used to index a vector of separate traces that will be merged into a single trace.

```
trace_vector: TYPE = [appl_id -> comp_trace]

merge(T: trace_vector, ids: id_list): RECURSIVE comp_trace =
   CASES ids OF
     null: null,
     cons(a, rest): IF T(a) = null
                      THEN cons(null_comp_event,
                                merge(T WITH [(a) := null], cdr(ids)))
                      ELSE cons(car(T(a)),
                                merge(T WITH [(a) := cdr(T(a))],
```

```
                                     cdr(ids)))
                      ENDIF
     ENDCASES
     MEASURE length(ids)
```

A little utility function to extract application IDs is needed as well.

```
  appl_ids(trace: comp_trace): RECURSIVE id_list =
     CASES trace OF
       null: null,
       cons(e, rest): cons(appl_id(e), appl_ids(rest))
     ENDCASES
     MEASURE length(trace)
```

As before, this could have been defined nonrecursively using the PVS primitive `map`, but proof considerations often favor a recursive definition.

Using these definitions, the alternative form of the verification result is as follows.

```
  well_partitioned: THEOREM
     proper_access(cmds) IMPLIES
        do_all(cmds) = merge(LAMBDA a: do_all(purge(cmds, a)),
                             appl_ids(do_all(cmds)))
```

This theorem has been shown to be a consequence of the `well_partitioned` result from Section 4.4.6.

4.5 ACR Design 2: Multiplexed Shared Resources

This section describes an extension to the PVS formalization of the baseline design in the previous section. The complete PVS theory can be found in Appendix B.1.

We assume a basic IMA architecture having the following characteristics:

- The system has a fixed set of applications.
- Several types of commands exist: machine instructions plus save and restore operations occurring at partition context switches.
- No interpartition communication is supported.
- Each resource is accessible by at most one application at a time, but some resources are shared and their values are swapped in and out during context switches.
- Resource allocation and access rights are dynamic for the shared resources and static for all others.

This configuration extends the baseline design in an important way. Although main memory may be statically allocated by the kernel, nearly every processor contains bits of state information such as register values that cannot be statically allocated, that are instead reallocated to the currently running partition on every context switch. This design handles such a feature by modeling a save area used to store the shared resources for all applications, presumably in the private memory of the kernel.

Command execution is required to follow the pattern of a sequence of instructions from the same application bracketed by matching restore and save operations. This discipline ensures that the instructions currently executing are allowed to access the shared resources. Save and restore operations themselves are not likely to be issued by the applications, but rather be introduced by the kernel as part of the context switch process. We include them to model the actions performed regardless of which entity initiates the actions.

4.5.1 Representation

Three main differences exist over the baseline design case: a set of shared resources is assumed, the system state has additional content, and there are new command types. The state now has three components: a "memory" area analogous to the previous concept of resource state, a save area to hold the values of shared resources, and the allocation state to record the dynamic assignment of access rights for resources.

```
shared_resources: resource_list

memory:       TYPE = [resource -> info]
save_area:    TYPE = [appl_id -> memory]
allocation:   TYPE = [resource -> access_set]
system_state: TYPE = [# active: memory, save: save_area,
                        alloc: allocation #]

cmd_type: TYPE = {INSTR, SAVE, RESTORE}
```

4.5.2 Computation

Command execution now has three types of commands to handle. The save and restore operations have two effects: moving shared resource values between active memory and the save area, and updating the allocation state of shared resources.

```
exec_save(m: memory, s: memory): memory =
   next_state(shared_resources, map(m)(shared_resources), s)

exec_restore(s: memory, m: memory): memory =
   next_state(shared_resources, map(s)(shared_resources), m)

execute(c: command, s: system_state): system_state =
   IF cmd_type(c) = INSTR
      THEN s WITH [(active) := exec_instr(c, active(s))]
   ELSIF cmd_type(c) = SAVE
      THEN s WITH [(save)(appl_id(c)) :=
                       exec_save(active(s), save(s)(appl_id(c))),
                     (alloc) := deallocate(alloc(s), appl_id(c))]
      ELSE s WITH [(active) :=
                       exec_restore(save(s)(appl_id(c)), active(s)),
                     (alloc) := allocate(alloc(s), appl_id(c))]
   ENDIF
```

Computation traces are constructed as before with the exception that not all commands are represented; save and restore commands are filtered out and do not become part of the trace. The rationale for this is that they are an artifact of the multiplexing of processor resources; they have no inherent meaning to the avionics functions of the applications.

```
do_all(cmds: cmd_list): RECURSIVE comp_trace =
   CASES cmds OF
     null: null,
     cons(c, rest):
        IF cmd_type(c) = INSTR
           THEN cons((# appl_id := appl_id(c),
                        cmd_type := cmd_type(c),
                        results := do_step(c, state(rest)) #),
                     do_all(rest))
           ELSE do_all(rest)
        ENDIF
   ENDCASES
   MEASURE length(cmds)
```

4.5.3 Separation

No new types or functions are needed to describe the separation of command stream execution.

4.5.4 Requirement

The partitioning requirement is expressed exactly as before.

4.5.5 Policy

The policy area has several new items. The `alloc` predicate is no longer a constant but has become part of the state. The initial state is a constant that must satisfy the allocation exclusivity condition.

```
exclusive(alloc: allocation): bool =
   FORALL (r: resource):
     EXISTS (a: appl_id):
       FORALL (ar: access_right):
         member(ar, alloc(r)) IMPLIES a = appl_id(ar)

initial_state: {s: system_state | exclusive(alloc(s))}
```

Functions are needed to define updates to the allocation state for save and restore commands. On a restore, the application is granted both read and write access to all shared resources. On a save, all access rights to the shared resources are rescinded.

```
shared_set(a: appl_id): access_set =
   add((# appl_id := a, mode := READ #),
       add((# appl_id := a, mode := WRITE #),
           emptyset[access_right]))
```

```
allocate(alloc: allocation, a: appl_id): allocation =
   LAMBDA (r: resource): IF member(r, shared_resources)
                            THEN shared_set(a)
                            ELSE alloc(r)
                         ENDIF

deallocate(alloc: allocation, a: appl_id): allocation =
   LAMBDA (r: resource): IF member(r, shared_resources)
                            THEN emptyset
                            ELSE alloc(r)
                         ENDIF
```

The basic conditions of the `proper_access` predicate remain the same except that the system state must now be consulted to obtain the allocation state.

```
proper_access(cmds: cmd_list): RECURSIVE bool =
   CASES cmds OF
     null: true,
     cons(c, rest):
       IF cmd_type(c) = INSTR
          THEN     mode_access(READ, args(c), appl_id(c), state(rest))
               AND mode_access(WRITE, results(c), appl_id(c), state(rest))
               AND proper_access(rest)
          ELSE proper_access(rest)
       ENDIF
   ENDCASES
   MEASURE length(cmds)
```

A significant addition to the baseline system is a well-formedness protocol on command sequencing. This is needed to enforce the requirement that instructions be bracketed by matching pairs of restore and save commands:

⟨ ... RESTORE, INSTR, ..., INSTR, SAVE ... ⟩

A `proper_swap` predicate is added for this purpose. It recursively checks the sequencing requirement for a given command list.

```
proper_swap_rec(cmds: cmd_list, active: bool,
                a: appl_id): RECURSIVE bool =
   CASES cmds OF
     null: NOT active,
     cons(c, rest): IF active AND a = appl_id(c)
                       THEN IF cmd_type(c) = RESTORE
                               THEN proper_swap_rec(rest, false,
                                                    default_appl)
                            ELSIF cmd_type(c) = INSTR
                               THEN proper_swap_rec(rest, true, a)
                               ELSE false
```

```
                           ENDIF
                  ELSIF NOT active AND cmd_type(c) = SAVE
                     THEN proper_swap_rec(rest, true, appl_id(c))
                     ELSE false
                  ENDIF
   ENDCASES
   MEASURE length(cmds)

 proper_swap(cmds: cmd_list): bool =
      proper_swap_rec(cmds, false, default_appl)
   OR (EXISTS (a: appl_id): proper_swap_rec(cmds, true, a))
```

Closely related to this predicate is a support function called `active_for`, which states when the tail of a command list leaves the system active for a given application. It follows a recursive trajectory similar to that of `proper_swap`.

```
 active_for(a: appl_id, cmds: cmd_list): RECURSIVE bool =
   CASES cmds OF
     null: false,
     cons(c, rest): IF cmd_type(c) = SAVE OR appl_id(c) /= a
                       THEN false
                    ELSIF cmd_type(c) = RESTORE
                       THEN true
                       ELSE active_for(a, rest)
                    ENDIF
   ENDCASES
   MEASURE length(cmds)
```

The complete validity condition on command lists now includes both the `proper_swap` and `proper_access` predicates.

```
 proper_commands(cmds: cmd_list): bool =
    proper_swap(cmds) AND proper_access(cmds)
```

4.5.6 Verification

The main partitioning theorem takes the same form as in the baseline system with the substitution of the `proper_commands` predicate for `proper_access`:

```
 well_partitioned: THEOREM
    proper_commands(cmds) IMPLIES
       purge(do_all(cmds), a) = do_all(purge(cmds, a))
```

This theorem has been proved using the PVS prover along with some 30 or so supporting lemmas. The proof involved some significantly more difficult twists due to the added complexity of the save and restore operations and the more complicated policy condition.

Several lemmas were needed to reason about the `active_for` concept, such as the following two:

```
active_unique: LEMMA
   active_for(a1, cmds) AND active_for(a2, cmds)
     IMPLIES a1 = a2

active_command: LEMMA
   proper_swap(cons(c, cmds)) IMPLIES
     IF cmd_type(c) = RESTORE
        THEN NOT active_for(appl_id(c), cmds)
        ELSE active_for(appl_id(c), cmds)
     ENDIF
```

The exclusivity condition now must be reestablished on every save and restore command. Moreover, relationships between save area values and active memory values must be expressed.

```
exclusive_allocate: LEMMA
   exclusive(alloc) IMPLIES exclusive(allocate(alloc, a))

execute_in_restore: LEMMA
     proper_commands(cmds) AND cmd_type(c) = RESTORE AND
     member(r, shared_resources)
   IMPLIES active(execute(c, state(cmds)))(r)
             = save(state(cmds))(appl_id(c))(r)
```

To aid the proof, an auxiliary concept was introduced called the `appl_state`. This function gives the value of a resource for a given application based on whether it should be found in active memory or in the save area. Using this notion, a modified state invariant was expressed and proved relating application states in the integrated and purged command cases.

```
appl_state(cmds, a)(r): info =
   IF active_for(a, cmds) OR NOT member(r, shared_resources)
      THEN active(state(cmds))(r)
      ELSE save(state(cmds))(a)(r)
   ENDIF

state_invariant: LEMMA
     proper_commands(cmds) AND
     (member(r, shared_resources) OR
       member((# appl_id := a, mode := READ #), alloc(state(cmds))(r)))
   IMPLIES
     appl_state(cmds, a)(r) = appl_state(purge(cmds, a), a)(r)
```

Establishing this invariant involves analyzing more cases than its counterpart in the baseline design, making the interactive PVS proof more cumbersome. This proof complexity is the cost of introducing resource sharing.

4.6 ACR Design 3: Interpartition Communication

This section describes a different extension to the PVS formalization of the baseline design, independent of the one in the previous section. The complete PVS theory can be found in Appendix C.1.

We assume a basic IMA architecture having the following characteristics:

- The system has a fixed set of applications.
- Two types of commands exist: machine instructions plus generic, interpartition communication (IPC) kernel services.
- Each resource is accessible by at most one application.
- Resource allocation and access rights are fully static.

This configuration extends the baseline design by adding the important feature of interpartition communication. The exact types of communication and specific kernel services for achieving them are not modeled. It suffices merely to allow for IPC commands that operate on a global IPC state while adhering to the same access control policy on resources that ordinary instructions observe. In fact, the kernel services are not limited to those used for IPC—most other types fit the model as well. Unlike design 2, command execution is not required to follow any patterns; the two types of commands can be interleaved as desired.

When IPC capability is added, the central problem that arises is that partitions are no longer noninterfering in the strict sense. Communicating applications do indeed "interfere" with one another. But this interdependence is intentional, of course, and we must accept the explicitly allowed interactions while prohibiting the unintended ones.

The primary means of achieving this goal is architectural. IPC is only allowed to occur through kernel services; no shared-memory communication is permitted. IPC services may cause updates to application-owned resources. Our model incorporates constraints sufficient to keep such updates confined to one partition at a time. The net result is that we can assure that third-party partitions are protected from unintended effects during IPC activity.

Modeling this arrangement requires additional mechanisms based on the introduction of global and local portions of the system state. Local states are replicated as before to capture the separate computation histories. A global state is added to capture the kernel's internal implementation of IPC. Each partition sees a common view of this IPC state.

Finally, note that even with the model presented in this section, the protection offered is limited to absence of harmful effects within application resources. It is still possible for mishandling to occur within the kernel before data has been delivered to its intended recipient. Chapter 5 will take up this problem, introducing a modeling technique to show that internal kernel behavior is also free from interfering effects.

4.6.1 Representation

The system state consists of two parts: the resource state holding application data, analogous to the state in design 1, and the IPC state, left uninterpreted in the model. IPC commands must access both parts of the system state while ordinary instructions access only the resource state.

```
res_state:    TYPE = [resource -> info]
IPC_state:    TYPE+
system_state: TYPE = [# res: res_state, IPC: IPC_state #]

initial_res_state: res_state
```

```
initial_IPC_state: IPC_state
initial_state:     system_state = (# res := initial_res_state,
                                     IPC := initial_IPC_state #)

cmd_type: TYPE = {INSTR, IPC}
```

4.6.2 Computation

Computation concepts for instructions are the same as before. For IPC commands, several new functions are introduced to model the effects of IPC kernel services. IPC command execution draws inputs from both the resource state and the IPC state, and likewise produces outputs for both. We are not concerned with the details of IPC state updates because we only wish to compare the results produced by all the applications. As long as the same effects occur in both system architectures, the exact nature of interpartition communication is immaterial.

```
exec_IPC(c: command, res: res_state, IPC: IPC_state): system_state =
   (# res := next_state(results(c),
                        res_update_IPC(c, map(res)(args(c)), IPC),
                        res),
      IPC := IPC_update_IPC(c, map(res)(args(c)), IPC) #)
```

Derivation of the current state from a command stream proceeds by accounting for the type of command executed at each step. This is a straightforward extension of the baseline model.

```
state(cmds: cmd_list): RECURSIVE system_state =
   CASES cmds OF
     null: initial_state,
     cons(c, rest): IF cmd_type(c) = INSTR
                       THEN (# res := execute(c, res(state(rest))),
                               IPC := IPC(state(rest)) #)
                       ELSE exec_IPC(c, res(state(rest)),
                                     IPC(state(rest)))
                    ENDIF
   ENDCASES
   MEASURE length(cmds)
```

An additional function represents the results computed during IPC command execution. The do_all function uses this function to construct the computation trace, which includes events for IPC commands.

```
do_IPC(c: command, res: res_state, IPC: IPC_state): info_list =
   res_update_IPC(c, map(res)(args(c)), IPC)

do_all(cmds: cmd_list): RECURSIVE comp_trace =
   CASES cmds OF
     null: null,
     cons(c, rest): cons(IF cmd_type(c) = INSTR
                            THEN INSTR_event(c, res(state(rest)))
```

```
                              ELSE IPC_event(c, res(state(rest)),
                                             IPC(state(rest)))
                           ENDIF,
                           do_all(rest))
    ENDCASES
    MEASURE length(cmds)
```

4.6.3 Separation

The concepts needed to describe processor separation are more complicated for this design. Federated system behavior cannot be modeled using n completely separate command streams, each operating independently of the others. Because of the IPC commands, computations among processors are interdependent. Results from one partition become inputs to another when IPC services are invoked. To handle this situation, we apply separation in a more selective manner.

First, we introduce a system state split into global and local parts. The kernel's internal state needed to implement IPC is modeled as a single, global state, common to all processors in the fictitious federated system. The separate resource states, one for each partition in the federated system model, are collected into a structure and referred to as local states. IPC commands operate on the global state and one of the local states. Conversely, instruction commands operate only on one of the local resource states.

This scheme requires a different approach to model the elaboration of computations. We begin with the types representing the foregoing state concepts. Local portions of the system state are accessed by indexing with application IDs. In addition to resource states, computation traces are kept within this structure. Traces are not part of the system state; it is simply convenient to keep a partition's trace together with its corresponding resource state.

```
trace_state_appl:   TYPE = [# trace: comp_trace, res: res_state #]
init_trace_state_appl: trace_state_appl =
                         (# trace := null, res := initial_res_state #)
trace_state_vector: TYPE = [appl_id -> trace_state_appl]
trace_state_full:   TYPE = [# local:  trace_state_vector,
                              global: IPC_state #]
```

It is also helpful to collect the local state and trace update expressions into a single update function.

```
comp_step(c: command, local: trace_state_appl,
          global: IPC_state): trace_state_appl =
   IF cmd_type(c) = IPC
      THEN (# trace := cons(IPC_event(c, res(local), global),
                            trace(local)),
              res   := res(exec_IPC(c, res(local), global)) #)
      ELSE (# trace := cons(INSTR_event(c, res(local)), trace(local)),
              res   := execute(c, res(local)) #)
   ENDIF
```

A command list is executed by the ensemble of separate processors and the common "kernel" that serves them. Each command updates the local state for one partition and the global IPC state in the case of IPC commands.

```
do_all_purge(cmds: cmd_list): RECURSIVE trace_state_full =
   CASES cmds OF
     null: (# local  := LAMBDA (a: appl_id): init_trace_state_appl,
              global := initial_IPC_state #),
     cons(c, rest):
         LET prev = do_all_purge(rest) IN
         (# local  :=
              LAMBDA (a: appl_id):
                 IF a = appl_id(c)
                    THEN comp_step(c, local(prev)(a), global(prev))
                    ELSE local(prev)(a)
                 ENDIF,
            global :=
              IF cmd_type(c) = IPC
                 THEN IPC(exec_IPC(c, res(local(prev)(appl_id(c))),
                                   global(prev)))
                 ELSE global(prev)
              ENDIF
          #)
   ENDCASES
   MEASURE length(cmds)
```

The function `do_all_purge` combines the roles previously served by the two functions `do_all` and `purge`. Two components are produced by this function: a vector of resource states and traces, one for each application, and a single, common IPC state. Execution of commands within `do_all_purge` keeps the partitions separate while allowing a common IPC state to evolve, thus ensuring that partitions receive meaningful values from their IPC operations, just as they do in the fully integrated system.

4.6.4 Requirement

The partitioning requirement is the same in spirit as that of the baseline model, but its expression is different owing to the way computation is modeled using the function `do_all_purge`.

```
purge(do_all(cmds), a) = trace(local(do_all_purge(cmds))(a))
```

The intent is to arrive at the separate computation traces as was done for the previous cases, while making allowances for the special circumstances surrounding IPC command execution. While the form of this requirement is not as elegant as that of Section 4.4.4, it serves the same purpose within the more challenging IPC context.

4.6.5 Policy

Access control policy in this design is identical to the baseline case. Each IPC command must adhere to the same access constraints as instruction commands. What this means is that an

IPC command may access only those resources assigned to the partition requesting the IPC service. This is a reasonable restriction, and the proof presented in the next section shows that it is sufficient to ensure strong partitioning.

4.6.6 Verification

Taking the partitioning requirement as expressed above, the main partitioning theorem for design 3 can be expressed as follows:

```
well_partitioned: THEOREM
   proper_access(cmds) IMPLIES
      purge(do_all(cmds), a) = trace(local(do_all_purge(cmds))(a))
```

This theorem has been proved in PVS with the help of some 20 supporting lemmas. The proof was more involved than the baseline case, but not overly so.

New lemmas introduced in this design were needed to deal with the effects of IPC command execution, such as the following lemma, which shows that if resource values match for a pair of states, then they still will match after executing an IPC command.

```
state_match(a, s1, s2): bool =
   FORALL r:
     member((# appl_id := a, mode := READ #), alloc(r))
       IMPLIES s1(r) = s2(r)

exec_IPC_match_appl: LEMMA
     mode_access(READ, args(c), a) AND
     state_match(a, s1, s2) AND a = appl_id(c)
   IMPLIES
     state_match(a, res(exec_IPC(c, s1, comm)),
                 res(exec_IPC(c, s2, comm)))
```

Several other lemmas needed to make deductions about IPC commands are shown below.

```
exec_IPC_IPC_appl: LEMMA
     mode_access(READ, args(c), a) AND
     state_match(a, s1, s2) AND a = appl_id(c)
   IMPLIES
     IPC(exec_IPC(c, s1, comm)) = IPC(exec_IPC(c, s2, comm))

exec_IPC_match_not_appl: LEMMA
     mode_access(WRITE, results(c), appl_id(c)) AND a /= appl_id(c)
   IMPLIES
     state_match(a, res(exec_IPC(c, s, comm)), s)

IPC_event_match_appl: LEMMA
     mode_access(READ, args(c), appl_id(c)) AND
     state_match(appl_id(c), s1, s2)
   IMPLIES
     IPC_event(c, s1, comm) = IPC_event(c, s2, comm)
```

Finally, the overall state invariant that applies after each command is shown below. The invariant asserts state matching conditions for both local and global state components.

```
state_invariant: THEOREM
   proper_access(cmds) IMPLIES
      (FORALL a: state_match(a, res(state(cmds)),
                             res(local(do_all_purge(cmds))(a)))) AND
      IPC(state(cmds)) = global(do_all_purge(cmds))
```

4.6.7 Shared Avionics Devices

In addition to IPC services, there is another area where applications may affect each other, namely, where external avionics devices are shared among multiple partitions. Allocation of such devices is typically dedicated rather than shared, but multiplexed access is a definite possibility in some architectures. For this reason, a partitioning model should accommodate this type of sharing if the need arises. We have not extended the core model of this report to cover this case, but we now sketch how the existing framework can support it.

Shared avionics devices can be handled in manner similar to IPC services. The model would need to recognize such I/O operations, whether carried out using kernel services or through direct access by machine instructions, as a special class and handle them accordingly. A treatment analogous to that of IPC services is the prescribed method. By making the special I/O operations adhere to the same constraints as IPC services, the same modeling and proof scheme can be used to show that the I/O has no effects outside of the designated resources. The global-plus-local state technique would work to perform the desired verification. Alternatively, a more general form of the model could be used in which IPC services and shared device I/O are both particular instances of the general class of shared operations requiring special treatment.

Chapter 5

Extending the Model to Kernel Partitioning

As discussed in the previous chapter, the resource partitioning models offer assurance against resource interference caused by other applications within a system. As long as a computation proceeds entirely within one partition, this property is sufficient to achieve independent operation. If, however, communication with other applications takes place, there are additional points of vulnerability. In particular, when data is in transit from one partition to another, temporarily being held within private ACR data structures rather than partition resources, there is a possibility of mishandling that is not covered by the previously developed models. We now turn our attention to this additional problem.

5.1 Kernel Noninterference

The basic approach we will follow is to apply the foregoing modeling framework and adapt it to the kernel interference problem. What this involves is taking the traditional noninterference concept and turning it upside down. Rather than separating the applications, we choose instead to separate the IPC mechanisms within the kernel. We assume the kernel implements IPC using conventional techniques such as ports or channels. Imagine that we can separate and replicate the kernel's processing, assigning each port or channel to its own kernel "machine." Then we apply the techniques of Section 4.6, inverting the roles of partitions and kernel. The partitions become the entity we hold constant while the kernel's IPC channels become the objects of separation as if implemented by a federated system.

Given this background sketch, the application of the kernel noninterference technique requires the following steps.

- Identify the virtual IPC structures implemented within the kernel, such as ports, channels, pipes, etc.
- Create a vector of local states for the kernel based on these IPC structures.
- Create a global state containing the partition resources.
- Model computation of regular machine instructions with respect to the common global state.

- Model computation of IPC services with respect to the particular local state corresponding to the designated port, channel, etc.
- Assert that the computation results of the integrated system are the same as those of the IPC-based federated system.

From the modeling standpoint, this scheme produces a valuable dual of the traditional noninterference structure, although it lacks some of its intuitive appeal. Moreover, the approach requires modeling more of the system design than is the case with resource partitioning. And it is important to note that no guarantee of correctness is inherent; the method only demonstrates that the IPC structures are independent. Nevertheless, the method does offer a tractable means of addressing the question of low-level interference within an ACR's operating system. We now give an illustration of the approach by inverting the model of Section 4.6.

5.2 Minimal Kernel Design for IPC

This model supplements the resource partitioning models by showing that interpartition communication through a kernel is not subject to unintended interference. A port-based IPC mechanism is included, and a simple set of IPC services (SEND and RECEIVE) is assumed. The complete PVS theory can be found in Appendix D.1.

We assume the basic IMA architecture of Section 4.6. The IPC commands are the two services SEND and RECEIVE. No restrictions are placed on connectivity; ports may connect two or more partitions. Ordinary queueing behavior within the virtual channels is observed. No bounds on the number of messages within a queue are specified. In practice such bounds may be necessary and will require a more elaborate model.

5.2.1 Representation

A port type is used to distinguish individual communication ports or channels implemented by the kernel.

```
port:  TYPE+
port:  [command -> port]
queue: TYPE = list[info_list]
```

The system state consists of two parts: the resource state holding application data, and the IPC state, which assigns one queue to each port. IPC commands must access both parts of the system state while ordinary instructions access only the resource state.

```
res_state:    TYPE = [resource -> info]
IPC_state:    TYPE = [port  -> queue]
system_state: TYPE = [# res: res_state, IPC: IPC_state #]

cmd_type: TYPE = {INSTR, SEND, RCV}
```

5.2.2 Computation

Computation concepts for instructions are the same as before. For IPC commands, several new functions are introduced to model the effects of the IPC services SEND and RCV.

```
do_SEND(c: command, res: res_state, IPC: IPC_state): info_list =
   map(res)(args(c))

do_RCV(c: command, res: res_state, IPC: IPC_state): info_list =
   IF IPC(port(c)) = null
      THEN null
      ELSE car(IPC(port(c)))
   ENDIF

exec_SEND(c: command, res: res_state, IPC: IPC_state): system_state =
   (# res := res,
      IPC := IPC WITH [(port(c)) :=
                          append(IPC(port(c)),
                                 (: do_SEND(c, res, IPC) :) )] #)

exec_RCV(c: command, res: res_state, IPC: IPC_state): system_state =
   (# res := next_state(results(c), do_RCV(c, res, IPC), res),
      IPC := IF IPC(port(c)) = null
                THEN IPC
                ELSE IPC WITH [(port(c)) := cdr(IPC(port(c)))]
             ENDIF #)

exec_IPC(c: command, res: res_state, IPC: IPC_state): system_state =
   IF cmd_type(c) = SEND
      THEN exec_SEND(c, res, IPC)
      ELSE exec_RCV(c, res, IPC)
   ENDIF
```

Derivation of the current state from a command stream proceeds by accounting for the type of command executed at each step. This uses basically the same functions as the previous model.

5.2.3 Separation

As before, the system state for the federated architecture is split into global and local parts. The partitions' resource state is modeled as a single, global state, common to all "processors" in the fictitious federated system. The kernel's internal state needed to implement IPC is separated into multiple copies, one for each port in the federated system model, and the set is collected into a structure and referred to as local states. IPC commands operate on the global state and one of the local states. Conversely, instruction commands operate only on the global state.

The elaboration of computation is inverted from the previous model, but otherwise works in the same manner. A composite structure containing the local and global states together with the computation trace is maintained. Only one computation trace, corresponding to the global resource state, is necessary.

```
IPC_state_vector: TYPE = [port -> IPC_state]
trace_state_full: TYPE = [# local:  IPC_state_vector,
                            global: res_state,
                            trace:  comp_trace #]
```

A command list is executed by the ensemble of partitions on one common processor and separate kernels for each port/channel. Each IPC command updates the kernel state for its port and the global resource state.

```
do_all_ports(cmds: cmd_list): RECURSIVE trace_state_full =
   CASES cmds OF
     null: (# local  := LAMBDA (p: port): initial_IPC_state,
              global := initial_res_state,
              trace  := null #),
     cons(c, rest):
         LET prev = do_all_ports(rest) IN
             IF cmd_type(c) = INSTR
                THEN (# local  := local(prev),
                        global := execute(c, global(prev)),
                        trace  := cons(INSTR_event(c, global(prev)),
                                       trace(prev)) #)
                ELSE (# local  :=
                          LAMBDA (p: port):
                            IF p = port(c)
                               THEN IPC(exec_IPC(c, global(prev),
                                                  local(prev)(p)))
                               ELSE local(prev)(p)
                          ENDIF,
                        global := res(exec_IPC(c, global(prev),
                                                local(prev)(port(c)))),
                        trace  := cons(IPC_event(c, global(prev),
                                                  local(prev)(port(c))),
                                       trace(prev)) #)
             ENDIF
   ENDCASES
   MEASURE length(cmds)
```

The function `do_all_ports` plays the same role as `do_all_purge` in the previous model. Three components are produced by this function: a vector of IPC states, one for each port, a single, common resource state, and a single computation trace. Execution of commands within `do_all_ports` keeps the IPC port structures separate while allowing a common resource state to evolve.

5.2.4 Requirement

The partitioning requirement is slightly different from that of the previous model, but its expression is simple due to the results produced by the function `do_all_ports`.

```
do_all(cmds) = trace(do_all_ports(cmds))
```

The intent is to assert that the computation results from the full set of applications is the same whether a single integrated kernel is used or an ensemble of port-separated kernels is used.

5.2.5 Policy

An access control policy is not necessary in this design because there is only a single thread of applications. In a sense the role of the policy is taken over by the design details introduced to model the IPC services.

5.2.6 Verification

Taking the partitioning requirement as expressed above, the main partitioning theorem for this design can be expressed as follows:

```
well_partitioned: THEOREM
      do_all(cmds) = trace(do_all_ports(cmds))
```

This theorem has been proved in PVS with the help of five supporting lemmas. The proof was simpler than that of the previous models, owing to the simple nature of the IPC mechanism employed.

The lemmas needed to make deductions about IPC commands are shown below.

```
IPC_event_match: LEMMA
    t1(port(c)) = t2(port(c)) IMPLIES
      IPC_event(c, s, t1) =  IPC_event(c, s, t2)

res_exec_IPC_match: LEMMA
    t1(port(c)) = t2(port(c)) IMPLIES
      res(exec_IPC(c, s, t1)) =  res(exec_IPC(c, s, t2))

IPC_exec_IPC_match: LEMMA
    p = port(c) AND t1(p) = t2(p) IMPLIES
      IPC(exec_IPC(c, s, t1))(p) = IPC(exec_IPC(c, s, t2))(p)

IPC_exec_IPC_other: LEMMA
    p /= port(c) IMPLIES IPC(exec_IPC(c, s, t))(p) = t(p)
```

The overall state invariant that applies after each command is shown below. This invariant asserts state matching conditions for both local and global state components.

```
state_invariant: THEOREM
      res(state(cmds)) = global(do_all_ports(cmds)) AND
      FORALL p: IPC(state(cmds))(p) =
                  local(do_all_ports(cmds))(p)(p)
```

Chapter 6

Conclusion

This report has presented a formal model of partitioning suitable for supporting an ACR architecture. Based in part on concepts drawn from the noninterference model used by researchers in information security, the model considers the way computations evolve in two different systems and requires that the results be equal. By defining what the system response should be in the case of a system of separate processors, the potentially interfering effects of integration can be assessed and identified.

The approach was demonstrated on three candidate designs, each an abstraction of features found in real systems. By continuing the development begun here, more realistic model instances can be constructed and used to represent more complex systems with a variety of architectural features and specific kernel services. The PVS notation was found to be effective in expressing the model, the key requirements, and the supporting lemmas. The PVS prover was found to be useful in carrying out the interactive proofs, all of which were completed for the designs undertaken.

Acknowledgments

The author is grateful for the cooperation and support of NASA Langley researchers during the course of this study, in particular, Ricky Butler. Discussions with Paul Miner of LaRC were also helpful in clarifying ideas. Participating in RTCA special committee SC-182 has been valuable in focusing on the important aspects of the avionics environment. The work of John Rushby from SRI International has likewise been valuable in identifying important issues related to partitioning. Ongoing support of the PVS toolset by SRI has kept the mechanical proof activity productive.

This work was supported in part by the National Aeronautics and Space Administration under Contract NAS1-96014.

Bibliography

[1] Aeronautical Radio, Inc., Annapolis, Maryland. *ARINC Specification 653: Avionics Application Software Standard Interface*, January 1997. Prepared by the Airlines Electronic Engineering Committee.

[2] Ricky W. Butler, James L. Caldwell, Victor A. Carreno, C. Michael Holloway, Paul S. Miner, and Ben L. Di Vito. NASA Langley's research and technology transfer program in formal methods. In *Tenth Annual Conference on Computer Assurance (COMPASS 95)*, Gaithersburg, MD, June 1995.

[3] Ricky W. Butler, Ben L. Di Vito, and C. Michael Holloway. Formal design and verification of a reliable computing platform for real-time control (Phase 3 results). NASA Technical Memorandum 109140, August 1994. Earlier reports are numbered 102716 and 104196.

[4] Joseph A. Goguen and José Meseguer. Security policies and security models. In *Proceedings of 1982 Symposium on Security and Privacy*, Oakland, California, May 1982. IEEE.

[5] Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *Proceedings of 1984 Symposium on Security and Privacy*, Oakland, California, May 1984. IEEE.

[6] J. Thomas Haigh and William D. Young. Extending the noninterference version of MLS for SAT. *IEEE Transations on Software Engineering*, 13(2):141–150, February 1987.

[7] National Aeronautics and Space Administration, Office of Safety and Mission Assurance, Washington, DC. *Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion*, July 1995.

[8] National Aeronautics and Space Administration, Office of Safety and Mission Assurance, Washington, DC. *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, Volume II: A Practitioner's Companion*, May 1997.

[9] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[10] Requirements and Technical Concepts for Aviation, Washington, DC. *Software Considerations in Airborne Systems and Equipment Certification*, December 1992. This document is known as EURO- CAE ED-12B in Europe.

[11] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, December 1992.

[12] John Rushby. Formal methods and digital systems validation for airborne systems. NASA Contractor Report 4551, December 1993.

[13] John Rushby. Formal methods and their role in digital systems validation for airborne systems. NASA Contractor Report 4673, August 1995.

Appendix A

Baseline Partitioning Model

The first model represents a baseline system having minimal features with fully static resource allocation and no interpartition communication.

A.1 PVS Theory

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%   Formal model of partitioning for an IMA architecture
%%
%%     Base partitioning model:
%%       - Basic system with fixed applications
%%       - Each resource is accessible by at most one application
%%       - Resource allocation and access rights are static
%%       - No interpartition communication
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


base_part_model: THEORY
BEGIN

% The system supports a fixed number of partitions (applications).

num_appl: posint

appl_id:  TYPE = below[num_appl]

% Basic information units are small (e.g., bytes)

info:      TYPE+
info_list: TYPE = list[info]
cmd_fn:    TYPE = [info_list -> info_list]

% Resources include memory locations, processor state, and some
% devices external to the ACR.

resource: TYPE+
```

```
resource_list: TYPE = list[resource]

% Commands include processor instructions and kernel services.  Input
% resources are listed in "args", output resources in "results".  The
% function models the operation performed on the arguments.

cmd_type: TYPE+
command:  TYPE = [# appl_id: appl_id, cmd_type: cmd_type,
                    args: resource_list, fn: cmd_fn,
                    results: resource_list #]
cmd_list: TYPE = list[command]

% Resource state models the memory available to applications

resource_state: TYPE = [resource -> info]
initial_state: resource_state

% Computation traces record the history of computed results

comp_event: TYPE = [# appl_id: appl_id, cmd_type: cmd_type,
                      results: info_list #]
comp_trace: TYPE = list[comp_event]


null_info: info
default_appl: appl_id
default_cmd: cmd_type
null_comp_event: comp_event =
   (# appl_id := default_appl, cmd_type := default_cmd, results := null #)


%%%%%%%%%%%%%%%%

% Access control on resources is modeled below.  Access control lists
% associated with each resource is the conceptual model of control.

access_mode:  TYPE = {READ, WRITE}
access_right: TYPE = [# appl_id: appl_id, mode: access_mode #]
access_set:   TYPE = set[access_right]

% The following asserts key requirements about resource allocation,
% namely, that it be static (independent of state) and exclusive (only
% one application has access rights to a resource).

allocation: TYPE = [resource -> access_set]

static_exclusive(alloc_fn: allocation): bool =
   FORALL (r: resource):
      EXISTS (a: appl_id):
         FORALL (ar: access_right):
           member(ar, alloc_fn(r)) IMPLIES a = appl_id(ar)
```

```
% Resource allocation is specified by the constant "alloc".

alloc: {p: allocation | static_exclusive(p)}


%%%%%%%%%%%%%%%%

% In the following, traces are represented in reverse chronological order,
% if read from left to right, owing to their use of the "list" data type.


% Update a list of resources within system memory with a list of values.

next_state(rlist: resource_list, values: info_list,
           s: resource_state): RECURSIVE resource_state =
   CASES rlist OF
     null: s,
     cons(r, rest): IF values = null
                      THEN next_state(rest, null, s) WITH [(r) := null_info]
                      ELSE next_state(rest, cdr(values), s)
                               WITH [(r) := car(values)]
                    ENDIF
   ENDCASES
   MEASURE length(rlist)

% The new state that results from executing a single instruction.

execute(c: command, s: resource_state): resource_state =
   next_state(results(c), fn(c)(map(s)(args(c))), s)

% The state results from the cumulative application of the entire
% command list.

state(cmds: cmd_list): RECURSIVE resource_state =
   CASES cmds OF
     null: initial_state,
     cons(c, rest): execute(c, state(rest))
   ENDCASES
   MEASURE length(cmds)

% do_step gives the values computed by a single command.

do_step(c: command, s: resource_state): info_list =
   fn(c)(map(s)(args(c)))

% Generate the full computation trace from the command sequence.

do_all(cmds: cmd_list): RECURSIVE comp_trace =
   CASES cmds OF
     null: null,
     cons(c, rest): cons((# appl_id := appl_id(c),
                            cmd_type := cmd_type(c),
```

```
                             results := do_step(c, state(rest)) #),
                          do_all(rest))
   ENDCASES
   MEASURE length(cmds)


%%%%%%%%%%%%%%%%

% Purge removes all commands/events not due to application a.

purge(cmds: cmd_list, a: appl_id): RECURSIVE cmd_list =
   CASES cmds OF
     null: null,
     cons(c, rest): IF a = appl_id(c)
                       THEN cons(c, purge(rest, a))
                       ELSE purge(rest, a)
                    ENDIF
   ENDCASES
   MEASURE length(cmds)

purge(trace: comp_trace, a: appl_id): RECURSIVE comp_trace =
   CASES trace OF
     null: null,
     cons(e, rest): IF a = appl_id(e)
                       THEN cons(e, purge(rest, a))
                       ELSE purge(rest, a)
                    ENDIF
   ENDCASES
   MEASURE length(trace)


%%%%%%%%%%%%%%%%

mode_access(m: access_mode, rlist: resource_list, a: appl_id): bool =
   FORALL (r: resource):
     member(r, rlist) IMPLIES
       member((# appl_id := a, mode := m #), alloc(r))

% This predicate formalizes the access rights needed to perform a command.
% A valid command list adheres to this condition, which corresponds to
% enforcing access control through runtime checks by the ACR.

proper_access(cmds: cmd_list): RECURSIVE bool =
   CASES cmds OF
     null: true,
     cons(c, rest): mode_access(READ, args(c), appl_id(c))
               AND  mode_access(WRITE, results(c), appl_id(c))
               AND  proper_access(rest)
   ENDCASES
   MEASURE length(cmds)
```

```
%%%%%%%%%%%%%%%%

a,b:    VAR appl_id
ar:     VAR access_right
c:      VAR command
cmds:   VAR cmd_list
ct:     VAR comp_trace
m:      VAR access_mode
r:      VAR resource
rlist:  VAR resource_list
s,s1,s2: VAR resource_state
vlist:  VAR info_list


% Utility lemmas

member_results: LEMMA
     member(r, results(c)) AND member(ar, alloc(r)) AND
     mode_access(WRITE, results(c), appl_id(c))
   IMPLIES appl_id(ar) = appl_id(c)

map_args: LEMMA
     (FORALL r: member(r, rlist) IMPLIES s1(r) = s2(r))
   IMPLIES
     map(s1)(rlist) = map(s2)(rlist)

next_state_not_in: LEMMA
   NOT member(r, rlist) IMPLIES next_state(rlist, vlist, s)(r) = s(r)

next_state_in: LEMMA
     member(r, rlist) IMPLIES
       next_state(rlist, vlist, s1)(r) = next_state(rlist, vlist, s2)(r)


% Lemmas on the effects of executing a single instruction.

execute_not_in: LEMMA
   NOT member(r, results(c)) IMPLIES execute(c, s)(r) = s(r)

execute_in: LEMMA
     (FORALL r: member(r, args(c)) IMPLIES s1(r) = s2(r))
   IMPLIES
     (FORALL r: member(r, results(c)) IMPLIES
                  execute(c, s1)(r) = execute(c, s2)(r))

execute_in_read: LEMMA
     member(r, results(c)) AND mode_access(READ, args(c), a) AND
     (FORALL r: member((# appl_id := a, mode := READ #), alloc(r))
                  IMPLIES s1(r) = s2(r))
   IMPLIES
     execute(c, s1)(r) = execute(c, s2)(r)
```

```
% Following is the key lemma relating ACR state values to those
% computed from the purged command streams.

state_invariant: LEMMA
     proper_access(cmds) AND
     member((# appl_id := a, mode := READ #), alloc(r))
   IMPLIES
     state(cmds)(r) = state(purge(cmds, a))(r)

purge_step: LEMMA
   proper_access(cons(c, cmds)) AND a = appl_id(c) IMPLIES
     do_step(c, state(cmds)) = do_step(c, state(purge(cmds, a)))

% This is the main result that asserts valid partitioning by showing
% that the purged operations produce the same
% outputs as the original integrated system.

well_partitioned: THEOREM
   proper_access(cmds) IMPLIES
      purge(do_all(cmds), a) = do_all(purge(cmds, a))

END base_part_model
```

A.2 Proof Summary

```
 Proof summary for theory base_part_model
    default_appl_TCC1.....................................proved - complete
    alloc_TCC1............................................proved - complete
    next_state_TCC1.......................................proved - complete
    next_state_TCC2.......................................proved - complete
    next_state_TCC3.......................................proved - complete
    state_TCC1............................................proved - complete
    purge_TCC1............................................proved - complete
    purge_TCC2............................................proved - complete
    member_results........................................proved - complete
    map_args..............................................proved - complete
    next_state_not_in.....................................proved - complete
    next_state_in.........................................proved - complete
    execute_not_in........................................proved - complete
    execute_in............................................proved - complete
    execute_in_read.......................................proved - complete
    state_invariant.......................................proved - complete
    purge_step............................................proved - complete
    well_partitioned......................................proved - complete
    Theory totals: 18 formulas, 18 attempted, 18 succeeded.
```

A.3 Proof Chain Analysis

```
base_part_model.well_partitioned has been PROVED.

  The proof chain for well_partitioned is COMPLETE.
```

```
well_partitioned depends on the following proved theorems:
  base_part_model.state_TCC1
  base_part_model.execute_not_in
  base_part_model.execute_in
  base_part_model.next_state_not_in
  integers.posint_TCC1
  integers.nonneg_int_TCC1
  base_part_model.purge_TCC2
  if_def.IF_TCC1
  base_part_model.next_state_TCC2
  list_props.length_TCC1
  base_part_model.purge_TCC1
  base_part_model.state_invariant
  base_part_model.next_state_TCC3
  list_props.member_TCC1
  integers.posint_TCC2
  base_part_model.purge_step
  base_part_model.map_args
  base_part_model.execute_in_read
  base_part_model.next_state_in
  base_part_model.member_results
  base_part_model.alloc_TCC1
  base_part_model.next_state_TCC1

well_partitioned depends on the following axioms:
  list_adt.list_induction

well_partitioned depends on the following definitions:
  base_part_model.purge
  base_part_model.proper_access
  sets.member
  base_part_model.execute
  list_adt.reduce_nat
  base_part_model.do_all
  list_props.length
  reals.>
  list_props.member
  list_adt_map.map
  reals.<=
  base_part_model.state
  sets.emptyset
  base_part_model.next_state
  reals.>=
  base_part_model.purge
  base_part_model.do_step
  base_part_model.mode_access
  base_part_model.static_exclusive
```

Appendix B

Shared Resource Model

The second design extends the baseline system in the first design by adding limited dynamic resource allocation. Shared resources are multiplexed by swapping their values in and out of memory.

B.1 PVS Theory

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%   Formal model of partitioning for an IMA architecture
%%
%%     Base model plus restricted resource sharing:
%%       - Basic system with fixed applications
%%       - Each resource is accessible by at most one application
%%       - Resource allocation and access rights are partly dynamic:
%%         some resources may be saved and restored during partition
%%         context switches
%%       - No interpartition communication
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


shared_part_model: THEORY
BEGIN

% The system supports a fixed number of partitions (applications).

num_appl: posint

appl_id:  TYPE = below[num_appl]

% Basic information units are small (e.g., bytes)

info:      TYPE+
info_list: TYPE = list[info]
cmd_fn:    TYPE = [info_list -> info_list]

% Resources include memory locations, processor state, devices external
```

```
% to the ACR.

resource: TYPE+
resource_list: TYPE = list[resource]

% Commands include processor instructions and kernel services.  Input
% resources are listed in "args", output resources in "results".  The
% function models the operation performed on the arguments.

cmd_type: TYPE = {INSTR, SAVE, RESTORE}
command:  TYPE = [# appl_id: appl_id, cmd_type: cmd_type,
                    args: resource_list, fn: cmd_fn,
                    results: resource_list #]
cmd_list: TYPE = list[command]

% Access control on resources is modeled below.  An access control list
% associated with each resource is the conceptual model of control.

access_mode:  TYPE = {READ, WRITE}
access_right: TYPE = [# appl_id: appl_id, mode: access_mode #]
access_set:   TYPE = set[access_right]

% System state models the memory available to applications and the save
% areas set aside for dynamically reallocated resources.

memory:       TYPE = [resource -> info]
save_area:    TYPE = [appl_id -> memory]
allocation:   TYPE = [resource -> access_set]
system_state: TYPE = [# active: memory, save: save_area, alloc: allocation #]

% Computation traces record the history of computed results

comp_event: TYPE = [# appl_id: appl_id, cmd_type: cmd_type,
                      results: info_list #]
comp_trace: TYPE = list[comp_event]

null_info: info
default_appl: appl_id
default_cmd: cmd_type
null_comp_event: comp_event =
   (# appl_id := default_appl, cmd_type := default_cmd, results := null #)


%%%%%%%%%%%%%%%%

% The following asserts what is required of resource allocation,
% namely, that it be exclusive (only one application has access
% rights to a resource).

exclusive(alloc: allocation): bool =
   FORALL (r: resource):
     EXISTS (a: appl_id):
```

```
      FORALL (ar: access_right):
        member(ar, alloc(r)) IMPLIES a = appl_id(ar)

initial_state: {s: system_state | exclusive(alloc(s))}

% Those resources that are shared (values saved and restored on swap)
% are indicated by this constant.

shared_resources: resource_list

% The dynamic resource allocation lists are updated using these functions.

shared_set(a: appl_id): access_set =
   add((# appl_id := a, mode := READ #),
          add((# appl_id := a, mode := WRITE #),
              emptyset[access_right]))

allocate(alloc: allocation, a: appl_id): allocation =
   LAMBDA (r: resource): IF member(r, shared_resources)
                            THEN shared_set(a)
                            ELSE alloc(r)
                         ENDIF

deallocate(alloc: allocation, a: appl_id): allocation =
   LAMBDA (r: resource): IF member(r, shared_resources)
                            THEN emptyset
                            ELSE alloc(r)
                         ENDIF


%%%%%%%%%%%%%%%%

% In the following, traces are represented in reverse chronological order,
% if read from left to right, owing to their use of the "list" data type.

% Update a list of resources within system memory with a list of values.

next_state(rlist: resource_list,
           values: info_list, s: memory): RECURSIVE memory =
   CASES rlist OF
     null: s,
     cons(r, rest): IF values = null
                      THEN next_state(rest, null, s) WITH [(r) := null_info]
                      ELSE next_state(rest, cdr(values), s)
                               WITH [(r) := car(values)]
                   ENDIF
   ENDCASES
   MEASURE length(rlist)

% The new state that results from executing a single instruction is
% computed below.
```

```
exec_instr(c: command, m: memory): memory =
   next_state(results(c), fn(c)(map(m)(args(c))), m)

exec_save(m: memory, s: memory): memory =
   next_state(shared_resources, map(m)(shared_resources), s)

exec_restore(s: memory, m: memory): memory =
   next_state(shared_resources, map(s)(shared_resources), m)

execute(c: command, s: system_state): system_state =
   IF cmd_type(c) = INSTR
      THEN s WITH [(active) := exec_instr(c, active(s))]
   ELSIF cmd_type(c) = SAVE
      THEN s WITH [(save)(appl_id(c)) :=
                       exec_save(active(s), save(s)(appl_id(c))),
                   (alloc) := deallocate(alloc(s), appl_id(c))]
      ELSE s WITH [(active) :=
                       exec_restore(save(s)(appl_id(c)), active(s)),
                   (alloc) := allocate(alloc(s), appl_id(c))]
   ENDIF

% The state results from the cumulative application of the entire
% command list.

state(cmds: cmd_list): RECURSIVE system_state =
   CASES cmds OF
     null: initial_state,
     cons(c, rest): execute(c, state(rest))
   ENDCASES
   MEASURE length(cmds)

% do_step gives the values computed by a single command.

do_step(c: command, s: system_state): info_list =
   fn(c)(map(active(s))(args(c)))

% Generate the full computation trace from the command sequence.

do_all(cmds: cmd_list): RECURSIVE comp_trace =
   CASES cmds OF
     null: null,
     cons(c, rest): IF cmd_type(c) = INSTR
                      THEN cons((# appl_id := appl_id(c),
                                   cmd_type := cmd_type(c),
                                   results := do_step(c, state(rest)) #),
                                do_all(rest))
                      ELSE do_all(rest)
                    ENDIF
   ENDCASES
   MEASURE length(cmds)

%%%%%%%%%%%%%%%%
```

```
% Purge removes all commands/events not issued by application a.

purge(cmds: cmd_list, a: appl_id): RECURSIVE cmd_list =
   CASES cmds OF
     null: null,
     cons(c, rest): IF a = appl_id(c)
                       THEN cons(c, purge(rest, a))
                       ELSE purge(rest, a)
                    ENDIF
   ENDCASES
   MEASURE length(cmds)

purge(trace: comp_trace, a: appl_id): RECURSIVE comp_trace =
   CASES trace OF
     null: null,
     cons(e, rest): IF a = appl_id(e)
                       THEN cons(e, purge(rest, a))
                       ELSE purge(rest, a)
                    ENDIF
   ENDCASES
   MEASURE length(trace)

%%%%%%%%%%%%%%%%

% The following predicate indicates whether an application is active
% after a command list is executed.  The condition holds when the last
% restore and all intervening instructions have the same application ID,
% and the matching save command has not yet occurred.

active_for(a: appl_id, cmds: cmd_list): RECURSIVE bool =
   CASES cmds OF
     null: false,
     cons(c, rest): IF cmd_type(c) = SAVE OR appl_id(c) /= a
                       THEN false
                    ELSIF cmd_type(c) = RESTORE
                       THEN true
                       ELSE active_for(a, rest)
                    ENDIF
   ENDCASES
   MEASURE length(cmds)

% Commands need to observe certain constraints for partition swapping
% to make sense.  The command list needs to be divisible into segments of
% the form <RESTORE,INSTR,...,INSTR,SAVE>, all with matching IDs.

proper_swap_rec(cmds: cmd_list, active: bool, a: appl_id): RECURSIVE bool =
   CASES cmds OF
     null: NOT active,
     cons(c, rest): IF active AND a = appl_id(c)
                       THEN IF cmd_type(c) = RESTORE
                               THEN proper_swap_rec(rest, false, default_appl)
```

```
                             ELSIF cmd_type(c) = INSTR
                                THEN proper_swap_rec(rest, true, a)
                                ELSE false
                             ENDIF
                     ELSIF NOT active AND cmd_type(c) = SAVE
                        THEN proper_swap_rec(rest, true, appl_id(c))
                        ELSE false
                     ENDIF
   ENDCASES
   MEASURE length(cmds)

proper_swap(cmds: cmd_list): bool =
      proper_swap_rec(cmds, false, default_appl)
   OR (EXISTS (a: appl_id): proper_swap_rec(cmds, true, a))

% "proper_access" formalizes the access rights needed to perform a command.

mode_access(m: access_mode, rlist: resource_list,
            a: appl_id, s: system_state): bool =
   FORALL (r: resource):
     member(r, rlist) IMPLIES
       member((# appl_id := a, mode := m #), alloc(s)(r))

proper_access(cmds: cmd_list): RECURSIVE bool =
   CASES cmds OF
     null: true,
     cons(c, rest):
       IF cmd_type(c) = INSTR
          THEN     mode_access(READ, args(c), appl_id(c), state(rest))
               AND mode_access(WRITE, results(c), appl_id(c), state(rest))
               AND proper_access(rest)
          ELSE proper_access(rest)
       ENDIF
   ENDCASES
   MEASURE length(cmds)

% "proper_commands" collects all the constraints on valid command lists.

proper_commands(cmds: cmd_list): bool =
   proper_swap(cmds) AND proper_access(cmds)


%%%%%%%%%%%%%%%%

a,a1,a2: VAR appl_id
alloc:  VAR allocation
ar:     VAR access_right
c:      VAR command
cmds,c1,c2:   VAR cmd_list
ct:     VAR comp_trace
m:      VAR access_mode
m1,m2:  VAR memory
```

```
p,q:    VAR bool
r:      VAR resource
rlist:  VAR resource_list
s,s1,s2: VAR system_state
vlist:  VAR info_list


% Utility lemmas

static_alloc_initial: LEMMA
   NOT member(r, shared_resources)
     IMPLIES alloc(state(cmds))(r) = alloc(initial_state)(r)

static_alloc: LEMMA
   NOT member(r, shared_resources)
     IMPLIES alloc(state(c1))(r) = alloc(state(c2))(r)

access_state: LEMMA
     (FORALL r: member(r, rlist) IMPLIES NOT member(r, shared_resources))
   IMPLIES
     mode_access(m, rlist, a, state(c1)) = mode_access(m, rlist, a, state(c2))

alloc_state_static: LEMMA
   member(ar, alloc(state(c1))(r)) AND NOT member(r, shared_resources)
     IMPLIES member(ar, alloc(state(c2))(r))

alloc_state_cons: LEMMA
   member(ar, alloc(state(cons(c, cmds)))(r)) AND cmd_type(c) = INSTR
     IMPLIES member(ar, alloc(state(cmds))(r))

member_allocate: LEMMA
   member(r, shared_resources)
     IMPLIES member((# appl_id := a, mode := READ #), allocate(alloc, a)(r))
         AND member((# appl_id := a, mode := WRITE #), allocate(alloc, a)(r))

member_appl_id: LEMMA
     member(r, rlist) AND member(ar, alloc(s)(r)) AND
     mode_access(m, rlist, a, s) AND exclusive(alloc(s))
   IMPLIES a = appl_id(ar)

map_args: LEMMA
     (FORALL r: member(r, rlist) IMPLIES m1(r) = m2(r))
   IMPLIES
     map(m1)(rlist) = map(m2)(rlist)

next_state_not_in: LEMMA
   NOT member(r, rlist) IMPLIES next_state(rlist, vlist, m1)(r) = m1(r)

next_state_in: LEMMA
     member(r, rlist) IMPLIES
       next_state(rlist, vlist, m1)(r) = next_state(rlist, vlist, m2)(r)
```

```
next_state_map: LEMMA
   member(r, rlist)
     IMPLIES next_state(rlist, map(m1)(rlist), m2)(r) = m1(r)

next_state_from: LEMMA
     member(r, rlist) AND length(rlist) = length(vlist) AND
       (FORALL (n: below[length(rlist)]): nth(vlist, n) = m1(nth(rlist, n)))
   IMPLIES next_state(rlist, vlist, m1)(r) = m1(r)

% Lemmas concerning the well formedness of command streams

active_unique: LEMMA
   active_for(a1, cmds) AND active_for(a2, cmds)
     IMPLIES a1 = a2

active_swap: LEMMA
   proper_swap_rec(cmds, true, a) IMPLIES active_for(a, cmds)

not_active_swap: LEMMA
   proper_swap_rec(cmds, false, a1)
     IMPLIES FORALL a2: NOT active_for(a2, cmds)

active_instr: LEMMA
   proper_swap(cons(c, cmds)) AND cmd_type(c) = INSTR
     IMPLIES active_for(appl_id(c), cmds)

active_save: LEMMA
   proper_swap(cons(c, cmds)) AND cmd_type(c) = SAVE
     IMPLIES active_for(appl_id(c), cmds)

active_restore: LEMMA
   proper_swap(cons(c, cmds)) AND cmd_type(c) = RESTORE
     IMPLIES NOT active_for(appl_id(c), cmds)

active_command: LEMMA
   proper_swap(cons(c, cmds)) IMPLIES
     IF cmd_type(c) = RESTORE
        THEN NOT active_for(appl_id(c), cmds)
        ELSE active_for(appl_id(c), cmds)
     ENDIF

other_not_active: LEMMA
   proper_swap(cons(c, cmds)) AND appl_id(c) /= a
     IMPLIES NOT active_for(a, cmds)

active_access: LEMMA
   active_for(a, cmds)
     IMPLIES mode_access(READ, shared_resources, a, state(cmds)) AND
             mode_access(WRITE, shared_resources, a, state(cmds))

exclusive_allocate: LEMMA
   exclusive(alloc) IMPLIES exclusive(allocate(alloc, a))
```

```
exclusive_deallocate: LEMMA
   exclusive(alloc) IMPLIES exclusive(deallocate(alloc, a))

exclusive_alloc: LEMMA
   exclusive(alloc(state(cmds)))

proper_swap_cons: LEMMA
   proper_swap(cons(c, cmds)) IMPLIES proper_swap(cmds)

proper_cons: LEMMA
   proper_commands(cons(c, cmds)) IMPLIES proper_commands(cmds)

active_purge: LEMMA
   proper_swap(cmds)
     IMPLIES active_for(a, cmds) = active_for(a, purge(cmds, a))

active_purge_equal: LEMMA
   proper_swap(cmds) AND active_for(a1, purge(cmds, a2))
     IMPLIES a1 = a2

proper_purge: LEMMA
   proper_commands(cmds) IMPLIES proper_commands(purge(cmds, a))

% Lemmas on the effects of executing a single instruction.

execute_not_in: LEMMA
   cmd_type(c) = INSTR AND NOT member(r, results(c))
     IMPLIES active(execute(c, s))(r) = active(s)(r)

execute_not_in_save: LEMMA
   cmd_type(c) = SAVE
     IMPLIES active(execute(c, s))(r) = active(s)(r)

execute_not_in_restore: LEMMA
   cmd_type(c) = RESTORE AND NOT member(r, shared_resources)
     IMPLIES active(execute(c, s))(r) = active(s)(r)

execute_save: LEMMA
   appl_id(c) /= a
     IMPLIES save(execute(c, state(cmds)))(a) = save(state(cmds))(a)

execute_in: LEMMA
     cmd_type(c) = INSTR AND
     (FORALL r: member(r, args(c)) IMPLIES active(s1)(r) = active(s2)(r))
   IMPLIES
     (FORALL r: member(r, results(c)) IMPLIES
                  active(execute(c, s1))(r) = active(execute(c, s2))(r))

execute_in_read: LEMMA
     cmd_type(c) = INSTR AND
     member(r, results(c)) AND mode_access(READ, args(c), a, s1) AND
```

```
    (FORALL r: member((# appl_id := a, mode := READ #), alloc(s1)(r))
                 IMPLIES active(s1)(r) = active(s2)(r))
   IMPLIES
    active(execute(c, s1))(r) = active(execute(c, s2))(r)

execute_in_save: LEMMA
    proper_commands(cmds) AND cmd_type(c) = SAVE AND
    member(r, shared_resources)
   IMPLIES save(execute(c, state(cmds)))(appl_id(c))(r)
            = active(state(cmds))(r)

execute_in_restore: LEMMA
    proper_commands(cmds) AND cmd_type(c) = RESTORE AND
    member(r, shared_resources)
   IMPLIES active(execute(c, state(cmds)))(r)
            = save(state(cmds))(appl_id(c))(r)

% Following is the key lemma relating ACR state values to those
% computed from the purged command streams.

appl_state(cmds, a)(r): info =
   IF active_for(a, cmds) OR NOT member(r, shared_resources)
     THEN active(state(cmds))(r)
     ELSE save(state(cmds))(a)(r)
   ENDIF

state_invariant: LEMMA
    proper_commands(cmds) AND
    (member(r, shared_resources) OR
      member((# appl_id := a, mode := READ #), alloc(state(cmds))(r)))
   IMPLIES
    appl_state(cmds, a)(r) = appl_state(purge(cmds, a), a)(r)

purge_step: LEMMA
   proper_commands(cons(c, cmds)) AND
   a = appl_id(c) AND cmd_type(c) = INSTR
    IMPLIES do_step(c, state(cmds)) = do_step(c, state(purge(cmds, a)))

% This is the main result that asserts valid partitioning by showing
% that the purged operations generate traces having the same outputs
% as the original integrated system.

well_partitioned: THEOREM
   proper_commands(cmds) IMPLIES
     purge(do_all(cmds), a) = do_all(purge(cmds, a))

END shared_part_model
```

B.2 Proof Summary

```
 Proof summary for theory shared_part_model
    default_appl_TCC1.....................................proved - complete
```

```
initial_state_TCC1.....................................proved - complete
next_state_TCC1........................................proved - complete
next_state_TCC2........................................proved - complete
next_state_TCC3........................................proved - complete
state_TCC1.............................................proved - complete
purge_TCC1.............................................proved - complete
purge_TCC2.............................................proved - complete
static_alloc_initial...................................proved - complete
static_alloc...........................................proved - complete
access_state...........................................proved - complete
alloc_state_static.....................................proved - complete
alloc_state_cons.......................................proved - complete
member_allocate........................................proved - complete
member_appl_id.........................................proved - complete
map_args...............................................proved - complete
next_state_not_in......................................proved - complete
next_state_in..........................................proved - complete
next_state_map.........................................proved - complete
next_state_from_TCC1...................................proved - complete
next_state_from........................................proved - complete
active_unique..........................................proved - complete
active_swap............................................proved - complete
not_active_swap........................................proved - complete
active_instr...........................................proved - complete
active_save............................................proved - complete
active_restore.........................................proved - complete
active_command.........................................proved - complete
other_not_active.......................................proved - complete
active_access..........................................proved - complete
exclusive_allocate.....................................proved - complete
exclusive_deallocate...................................proved - complete
exclusive_alloc........................................proved - complete
proper_swap_cons.......................................proved - complete
proper_cons............................................proved - complete
active_purge...........................................proved - complete
active_purge_equal.....................................proved - complete
proper_purge...........................................proved - complete
execute_not_in.........................................proved - complete
execute_not_in_save....................................proved - complete
execute_not_in_restore.................................proved - complete
execute_save...........................................proved - complete
execute_in.............................................proved - complete
execute_in_read........................................proved - complete
execute_in_save........................................proved - complete
execute_in_restore.....................................proved - complete
state_invariant........................................proved - complete
purge_step.............................................proved - complete
well_partitioned.......................................proved - complete
Theory totals: 49 formulas, 49 attempted, 49 succeeded.
```

B.3 Proof Chain Analysis

```
shared_part_model.well_partitioned has been PROVED.

  The proof chain for well_partitioned is COMPLETE.

  well_partitioned depends on the following proved theorems:
    shared_part_model.active_purge
    if_def.IF_TCC1
    shared_part_model.active_swap
    shared_part_model.next_state_not_in
    shared_part_model.exclusive_alloc
    integers.posint_TCC1
    shared_part_model.static_alloc_initial
    shared_part_model.next_state_TCC3
    shared_part_model.purge_step
    shared_part_model.active_restore
    shared_part_model.execute_in_restore
    shared_part_model.purge_TCC2
    shared_part_model.proper_swap_cons
    shared_part_model.execute_not_in
    shared_part_model.static_alloc
    shared_part_model.next_state_TCC2
    list_props.length_TCC1
    shared_part_model.execute_in_read
    shared_part_model.map_args
    shared_part_model.proper_cons
    shared_part_model.execute_not_in_save
    shared_part_model.member_allocate
    integers.posint_TCC2
    shared_part_model.active_purge_equal
    shared_part_model.active_access
    shared_part_model.next_state_in
    shared_part_model.default_appl_TCC1
    shared_part_model.state_invariant
    shared_part_model.purge_TCC1
    shared_part_model.proper_purge
    shared_part_model.execute_save
    shared_part_model.active_unique
    shared_part_model.exclusive_allocate
    shared_part_model.other_not_active
    shared_part_model.next_state_TCC1
    shared_part_model.active_command
    list_props.member_TCC1
    shared_part_model.next_state_map
    shared_part_model.not_active_swap
    shared_part_model.execute_in_save
    shared_part_model.exclusive_deallocate
    shared_part_model.execute_not_in_restore
    shared_part_model.active_instr
    shared_part_model.active_save
    integers.nonneg_int_TCC1
    shared_part_model.execute_in
```

```
  shared_part_model.initial_state_TCC1
  shared_part_model.state_TCC1

well_partitioned depends on the following axioms:
  list_adt.list_induction

well_partitioned depends on the following definitions:
  shared_part_model.exec_restore
  shared_part_model.mode_access
  list_adt.reduce_nat
  shared_part_model.exec_save
  notequal./=
  shared_part_model.proper_access
  shared_part_model.proper_swap
  shared_part_model.exec_instr
  reals.<=
  shared_part_model.active_for
  shared_part_model.proper_commands
  shared_part_model.purge
  shared_part_model.allocate
  sets.emptyset
  reals.>
  shared_part_model.do_all
  shared_part_model.exclusive
  shared_part_model.do_step
  sets.member
  shared_part_model.appl_state
  shared_part_model.state
  list_props.length
  shared_part_model.next_state
  shared_part_model.execute
  list_props.member
  shared_part_model.purge
  shared_part_model.deallocate
  list_adt_map.map
  shared_part_model.shared_set
  shared_part_model.proper_swap_rec
  reals.>=
  sets.add
```

Appendix C

IPC Partitioning Model

The third design extends the baseline system in a different way by allowing generic interpartition communication (IPC) services.

C.1 PVS Theory

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%   Formal model of partitioning for an IMA architecture
%%
%%     Base model plus interpartition communication:
%%       - Basic system with fixed applications
%%       - Each writable resource is accessible by at most
%%         one application
%%       - Read-only resources accessible by any application
%%       - Resource allocation and access rights are static
%%       - Generic interpartition communication is allowed
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


IPC_part_model: THEORY
BEGIN

% The system supports a fixed number of partitions (applications).

num_appl: posint

appl_id:  TYPE = below[num_appl]

% Basic information units are small (e.g., bytes)

info:      TYPE+
info_list: TYPE = list[info]
cmd_fn:    TYPE = [info_list -> info_list]

% Resources include memory locations, processor state, and some
% devices external to the ACR.
```

```
resource: TYPE+
resource_list: TYPE = list[resource]

% Commands include processor instructions and kernel services.  Input
% resources are listed in "args", output resources in "results".  The
% function models the operation performed on the arguments.

cmd_type: TYPE = {INSTR, IPC}
command:  TYPE = [# appl_id: appl_id, cmd_type: cmd_type,
                    args: resource_list, fn: cmd_fn,
                    results: resource_list #]
cmd_list: TYPE = list[command]

% The type res_state models the memory-like resources available to
% applications, IPC_state models the interpartition communication
% state, and system_state contains them both.

res_state:    TYPE = [resource -> info]
IPC_state:    TYPE+
system_state: TYPE = [# res: res_state, IPC: IPC_state #]

initial_res_state: res_state
initial_IPC_state: IPC_state
initial_state:     system_state = (# res := initial_res_state,
                                     IPC := initial_IPC_state #)

% Computation traces record the history of computed results

comp_event: TYPE = [# appl_id: appl_id, cmd_type: cmd_type,
                      args: resource_list, fn: cmd_fn,
                      res_res: resource_list,
                      results: info_list #]
comp_trace: TYPE = list[comp_event]


% Misc. constants

null_info: info
default_appl: appl_id
default_cmd: cmd_type
id_fn: cmd_fn = (LAMBDA (L: info_list): L)

null_comp_event: comp_event =
   (# appl_id := default_appl, cmd_type := default_cmd,
      args := null, fn := id_fn, res_res := null,
      results := null #)


%%%%%%%%%%%%%%%%

% Access control on resources is modeled below.  Access control lists
```

```
% associated with each resource is the conceptual model of control.

access_mode:  TYPE = {READ, WRITE}
access_right: TYPE = [# appl_id: appl_id, mode: access_mode #]
access_set:   TYPE = set[access_right]

read_only: [resource -> bool]

% The following asserts key requirements about static resource allocation,
% namely, that it be exclusive (only one application has access rights
% to a resource).

allocation: TYPE = [resource -> access_set]

static_exclusive(alloc_fn: allocation): bool =
   FORALL (r: resource):
     NOT read_only(r) IMPLIES
       EXISTS (a: appl_id):
         FORALL (ar: access_right):
           member(ar, alloc_fn(r)) IMPLIES a = appl_id(ar)

write_limited(alloc_fn: allocation): bool =
   FORALL (r: resource), (ar: access_right):
     read_only(r) AND member(ar, alloc_fn(r)) IMPLIES mode(ar) = READ

% Resource allocation is specified by the constant "alloc".

alloc: {p: allocation | static_exclusive(p) AND write_limited(p)}


%%%%%%%%%%%%%%%%

% In the following, traces are represented in reverse chronological order,
% if read from left to right, owing to their use of the "list" data type.


% Update a list of resources within system memory with a list of values.

next_state(rlist: resource_list, values: info_list,
           s: res_state): RECURSIVE res_state =
   CASES rlist OF
     null: s,
     cons(r, rest): IF values = null
                      THEN next_state(rest, null, s) WITH [(r) := null_info]
                      ELSE next_state(rest, cdr(values), s)
                               WITH [(r) := car(values)]
                    ENDIF
   ENDCASES
   MEASURE length(rlist)


% The new state that results from executing a single instruction.
```

```
execute(c: command, s: res_state): res_state =
   next_state(results(c), fn(c)(map(s)(args(c))), s)

% do_step gives the values computed by a single instruction.

do_step(c: command, s: res_state): info_list =
   fn(c)(map(s)(args(c)))


% Execution of IPC commands is modeled by the following.

res_update_IPC(c: command, args: info_list, IPC: IPC_state): info_list

IPC_update_IPC(c: command, args: info_list, IPC: IPC_state): IPC_state

exec_IPC(c: command, res: res_state, IPC: IPC_state): system_state =
   (# res := next_state(results(c),
                        res_update_IPC(c, map(res)(args(c)), IPC),
                        res),
      IPC := IPC_update_IPC(c, map(res)(args(c)), IPC) #)

% do_IPC gives the values computed by a single IPC command.

do_IPC(c: command, res: res_state, IPC: IPC_state): info_list =
   res_update_IPC(c, map(res)(args(c)), IPC)


% A system state results from the cumulative application of an entire
% command list, or from a command list segment continuing from a
% previously obtained state.

state(cmds: cmd_list): RECURSIVE system_state =
   CASES cmds OF
     null: initial_state,
     cons(c, rest): IF cmd_type(c) = INSTR
                       THEN (# res := execute(c, res(state(rest))),
                               IPC := IPC(state(rest)) #)
                       ELSE exec_IPC(c, res(state(rest)),
                                     IPC(state(rest)))
                    ENDIF
   ENDCASES
   MEASURE length(cmds)


% Construction functions for trace events

INSTR_event(c: command, res: res_state): comp_event =
   (# appl_id  := appl_id(c),
      cmd_type := cmd_type(c),
      args     := args(c),
      fn       := fn(c),
```

```
      res_res  := results(c),
      results  := do_step(c, res) #)

IPC_event(c: command, res: res_state, IPC: IPC_state): comp_event =
   (# appl_id  := appl_id(c),
      cmd_type := cmd_type(c),
      args     := args(c),
      fn       := fn(c),
      res_res  := results(c),
      results  := do_IPC(c, res, IPC) #)

% Generate the full computation trace from the command sequence.

do_all(cmds: cmd_list): RECURSIVE comp_trace =
   CASES cmds OF
     null: null,
     cons(c, rest): cons(IF cmd_type(c) = INSTR
                            THEN INSTR_event(c, res(state(rest)))
                            ELSE IPC_event(c, res(state(rest)),
                                                IPC(state(rest)))
                         ENDIF,
                         do_all(rest))
   ENDCASES
   MEASURE length(cmds)


%%%%%%%%%%%%%%%%

% Purge removes all events not due to application a.

purge(trace: comp_trace, a: appl_id): RECURSIVE comp_trace =
   CASES trace OF
     null: null,
     cons(e, rest): IF a = appl_id(e)
                       THEN cons(e, purge(rest, a))
                       ELSE purge(rest, a)
                    ENDIF
   ENDCASES
   MEASURE length(trace)


%%%%%%%%%%%%%%%%

% Computation using separated command streams involves computing both
% local and global state values.  The IPC state is assumed to be
% global (single-thread).

trace_state_appl:   TYPE = [# trace: comp_trace, res: res_state #]
init_trace_state_appl: trace_state_appl =
                         (# trace := null, res := initial_res_state #)
trace_state_vector: TYPE = [appl_id -> trace_state_appl]
trace_state_full:   TYPE = [# local:  trace_state_vector,
```

```
                              global: IPC_state #]

% A single command updates the relevant state and adds to the cumulative
% computation trace.

comp_step(c: command, local: trace_state_appl,
          global: IPC_state): trace_state_appl =
   IF cmd_type(c) = IPC
      THEN (# trace := cons(IPC_event(c, res(local), global), trace(local)),
              res   := res(exec_IPC(c, res(local), global)) #)
      ELSE (# trace := cons(INSTR_event(c, res(local)), trace(local)),
              res   := execute(c, res(local)) #)
   ENDIF

% A command list is executed by the ensemble of separate processors.
% Each command updates the local state for one partition and the
% global IPC state for IPC commands.

do_all_purge(cmds: cmd_list): RECURSIVE trace_state_full =
   CASES cmds OF
     null: (# local  := LAMBDA (a: appl_id): init_trace_state_appl,
              global := initial_IPC_state #),
     cons(c, rest):
         LET prev = do_all_purge(rest) IN
         (# local  :=
              LAMBDA (a: appl_id):
                 IF a = appl_id(c)
                    THEN comp_step(c, local(prev)(a), global(prev))
                    ELSE local(prev)(a)
                 ENDIF,
            global := IF cmd_type(c) = IPC
                        THEN IPC(exec_IPC(c, res(local(prev)(appl_id(c))),
                                                  global(prev)))
                        ELSE global(prev)
                     ENDIF
          #)
   ENDCASES
   MEASURE length(cmds)


%%%%%%%%%%%%%%%%

mode_access(m: access_mode, rlist: resource_list, a: appl_id): bool =
   FORALL (r: resource):
     member(r, rlist) IMPLIES
       member((# appl_id := a, mode := m #), alloc(r))

% This predicate formalizes the access rights needed to perform a command.
% A valid command list adheres to this condition, which corresponds to
% enforcing access control through runtime checks by the ACR.

proper_access(cmds: cmd_list): RECURSIVE bool =
```

```
   CASES cmds OF
     null: true,
     cons(c, rest): mode_access(READ, args(c), appl_id(c))
               AND  mode_access(WRITE, results(c), appl_id(c))
               AND  proper_access(rest)
   ENDCASES
   MEASURE length(cmds)


%%%%%%%%%%%%%%%%

a,b:    VAR appl_id
ar:     VAR access_right
c,d:    VAR command
cmds,c1,c2: VAR cmd_list
comm:   VAR IPC_state
ct:     VAR comp_trace
init:   VAR system_state
m:      VAR access_mode
r:      VAR resource
rlist:  VAR resource_list
s,s1,s2: VAR res_state
vlist:  VAR info_list


% It is useful to consider when two states match on all resources that
% a partition has read access to.

state_match(a, s1, s2): bool =
   FORALL r:
     member((# appl_id := a, mode := READ #), alloc(r)) IMPLIES s1(r) = s2(r)


% Utility lemmas

state_match_trans: LEMMA
     state_match(a, s1, s) AND state_match(a, s, s2)
       IMPLIES state_match(a, s1, s2)

member_results: LEMMA
     member(r, results(c)) AND member(ar, alloc(r)) AND
     mode_access(WRITE, results(c), appl_id(c))
   IMPLIES appl_id(ar) = appl_id(c)

map_args: LEMMA
     (FORALL r: member(r, rlist) IMPLIES s1(r) = s2(r))
   IMPLIES
     map(s1)(rlist) = map(s2)(rlist)

next_state_not_in: LEMMA
   NOT member(r, rlist) IMPLIES next_state(rlist, vlist, s)(r) = s(r)
```

```
next_state_in: LEMMA
     member(r, rlist) IMPLIES
       next_state(rlist, vlist, s1)(r) = next_state(rlist, vlist, s2)(r)


% Lemmas on the effects of executing a single instruction.

execute_not_in: LEMMA
   NOT member(r, results(c)) IMPLIES execute(c, s)(r) = s(r)

execute_in: LEMMA
     (FORALL r: member(r, args(c)) IMPLIES s1(r) = s2(r))
   IMPLIES
     (FORALL r: member(r, results(c)) IMPLIES
                  execute(c, s1)(r) = execute(c, s2)(r))

execute_in_read: LEMMA
     member(r, results(c)) AND mode_access(READ, args(c), a) AND
     (FORALL r: member((# appl_id := a, mode := READ #), alloc(r))
                  IMPLIES s1(r) = s2(r))
   IMPLIES
     execute(c, s1)(r) = execute(c, s2)(r)

execute_match_appl: LEMMA
     mode_access(READ, args(c), appl_id(c)) AND
     state_match(appl_id(c), s1, s2)
   IMPLIES
     state_match(appl_id(c), execute(c, s1), execute(c, s2))

execute_match_not_appl: LEMMA
     mode_access(WRITE, results(c), appl_id(c)) AND a /= appl_id(c)
   IMPLIES
     state_match(a, execute(c, s), s)

INSTR_event_match_appl: LEMMA
     mode_access(READ, args(c), appl_id(c)) AND
     state_match(appl_id(c), s1, s2)
   IMPLIES
     INSTR_event(c, s1) = INSTR_event(c, s2)


% Lemmas on the effects of executing a single IPC kernel service.

exec_IPC_not_in: LEMMA
   NOT member(r, results(c)) IMPLIES res(exec_IPC(c, s, comm))(r) = s(r)

exec_IPC_in: LEMMA
     (FORALL r: member(r, args(c)) IMPLIES s1(r) = s2(r))
   IMPLIES
     (FORALL r: member(r, results(c)) IMPLIES
                  res(exec_IPC(c, s1, comm))(r) = res(exec_IPC(c, s2, comm))(r))
       AND IPC(exec_IPC(c, s1, comm)) = IPC(exec_IPC(c, s2, comm))
```

```
exec_IPC_in_read: LEMMA
     member(r, results(c)) AND mode_access(READ, args(c), a) AND
     (FORALL r: member((# appl_id := a, mode := READ #), alloc(r))
                  IMPLIES s1(r) = s2(r))
   IMPLIES
     res(exec_IPC(c, s1, comm))(r) = res(exec_IPC(c, s2, comm))(r)
       AND IPC(exec_IPC(c, s1, comm)) = IPC(exec_IPC(c, s2, comm))

exec_IPC_in_IPC: LEMMA
     (FORALL r: member(r, args(c)) IMPLIES s1(r) = s2(r))
   IMPLIES
     IPC(exec_IPC(c, s1, comm)) = IPC(exec_IPC(c, s2, comm))

exec_IPC_match_appl: LEMMA
     mode_access(READ, args(c), a) AND
     state_match(a, s1, s2) AND a = appl_id(c)
   IMPLIES
     state_match(a, res(exec_IPC(c, s1, comm)), res(exec_IPC(c, s2, comm)))

exec_IPC_IPC_appl: LEMMA
     mode_access(READ, args(c), a) AND
     state_match(a, s1, s2) AND a = appl_id(c)
   IMPLIES
     IPC(exec_IPC(c, s1, comm)) = IPC(exec_IPC(c, s2, comm))

exec_IPC_match_not_appl: LEMMA
     mode_access(WRITE, results(c), appl_id(c)) AND a /= appl_id(c)
   IMPLIES
     state_match(a, res(exec_IPC(c, s, comm)), s)

IPC_event_match_appl: LEMMA
     mode_access(READ, args(c), appl_id(c)) AND
     state_match(appl_id(c), s1, s2)
   IMPLIES
     IPC_event(c, s1, comm) = IPC_event(c, s2, comm)


%%%%%%%%%%%%%%%

% Following are the key lemmas relating ACR state values to those
% computed from the purged command streams.

state_invariant: THEOREM
   proper_access(cmds) IMPLIES
      (FORALL a: state_match(a, res(state(cmds)),
                             res(local(do_all_purge(cmds))(a)))) AND
      IPC(state(cmds)) = global(do_all_purge(cmds))

well_partitioned: THEOREM
   proper_access(cmds) IMPLIES
      purge(do_all(cmds), a) = trace(local(do_all_purge(cmds))(a))
```

```
END IPC_part_model
```

C.2 Proof Summary

```
 Proof summary for theory IPC_part_model
    default_appl_TCC1.....................................proved - complete
    alloc_TCC1............................................proved - complete
    next_state_TCC1.......................................proved - complete
    next_state_TCC2.......................................proved - complete
    next_state_TCC3.......................................proved - complete
    state_TCC1............................................proved - complete
    state_TCC2............................................proved - complete
    do_all_TCC1...........................................proved - complete
    purge_TCC1............................................proved - complete
    purge_TCC2............................................proved - complete
    state_match_trans.....................................proved - complete
    member_results........................................proved - complete
    map_args..............................................proved - complete
    next_state_not_in.....................................proved - complete
    next_state_in.........................................proved - complete
    execute_not_in........................................proved - complete
    execute_in............................................proved - complete
    execute_in_read.......................................proved - complete
    execute_match_appl....................................proved - complete
    execute_match_not_appl................................proved - complete
    INSTR_event_match_appl................................proved - complete
    exec_IPC_not_in.......................................proved - complete
    exec_IPC_in...........................................proved - complete
    exec_IPC_in_read......................................proved - complete
    exec_IPC_in_IPC.......................................proved - complete
    exec_IPC_match_appl...................................proved - complete
    exec_IPC_IPC_appl.....................................proved - complete
    exec_IPC_match_not_appl...............................proved - complete
    IPC_event_match_appl..................................proved - complete
    state_invariant.......................................proved - complete
    well_partitioned......................................proved - complete
    Theory totals: 31 formulas, 31 attempted, 31 succeeded.
```

C.3 Proof Chain Analysis

```
IPC_part_model.well_partitioned has been PROVED.

  The proof chain for well_partitioned is COMPLETE.

  well_partitioned depends on the following proved theorems:
    IPC_part_model.next_state_in
    if_def.IF_TCC1
    IPC_part_model.do_all_TCC1
    IPC_part_model.state_invariant
```

```
  integers.posint_TCC1
  IPC_part_model.exec_IPC_in
  IPC_part_model.next_state_TCC3
  IPC_part_model.exec_IPC_match_not_appl
  list_props.length_TCC1
  IPC_part_model.next_state_TCC2
  IPC_part_model.state_match_trans
  IPC_part_model.execute_match_appl
  IPC_part_model.exec_IPC_IPC_appl
  integers.posint_TCC2
  IPC_part_model.execute_in
  IPC_part_model.alloc_TCC1
  IPC_part_model.purge_TCC2
  IPC_part_model.next_state_not_in
  IPC_part_model.INSTR_event_match_appl
  IPC_part_model.state_TCC1
  IPC_part_model.exec_IPC_in_IPC
  IPC_part_model.execute_match_not_appl
  IPC_part_model.IPC_event_match_appl
  list_props.member_TCC1
  IPC_part_model.member_results
  IPC_part_model.exec_IPC_match_appl
  IPC_part_model.execute_in_read
  IPC_part_model.next_state_TCC1
  IPC_part_model.execute_not_in
  IPC_part_model.purge_TCC1
  IPC_part_model.map_args
  integers.nonneg_int_TCC1
  IPC_part_model.state_TCC2
  IPC_part_model.exec_IPC_not_in
  IPC_part_model.exec_IPC_in_read

well_partitioned depends on the following axioms:
  list_adt.list_induction

well_partitioned depends on the following definitions:
  IPC_part_model.do_all_purge
  IPC_part_model.do_IPC
  IPC_part_model.state
  list_adt.reduce_nat
  IPC_part_model.initial_state
  IPC_part_model.comp_step
  IPC_part_model.do_all
  notequal./=
  IPC_part_model.proper_access
  reals.<=
  IPC_part_model.purge
  IPC_part_model.write_limited
  sets.emptyset
  IPC_part_model.do_step
  IPC_part_model.static_exclusive
  IPC_part_model.execute
```

```
IPC_part_model.init_trace_state_appl
IPC_part_model.mode_access
IPC_part_model.exec_IPC
reals.>
IPC_part_model.INSTR_event
sets.member
list_props.length
IPC_part_model.IPC_event
list_props.member
IPC_part_model.state_match
list_adt_map.map
reals.>=
IPC_part_model.next_state
```

Appendix D

Kernel Partitioning Model

This model supplements the resource partitioning models by showing that interpartition communication through a kernel is not subject to unintended interference. A simple set of IPC services (SEND and RECEIVE) is assumed.

D.1 PVS Theory

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%   Formal model of partitioning for an IMA architecture
%%
%%     Kernel (IPC) partitioning model:
%%       - Basic system with fixed applications
%%       - Limited interpartition communication (IPC) is allowed
%%       - Partitioning requirements apply to kernel state and IPC
%%         mechanism rather than resource state
%%       - Simple IPC services based on send, receive commands
%%       - Common resource state, multiple IPC states
%%       - Resource allocation and access rights irrelevant
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


kernel_part_model: THEORY
BEGIN

% The system supports a fixed number of partitions (applications).

num_appl: posint

appl_id:  TYPE = below[num_appl]

% Basic information units are small (e.g., bytes)

info:      TYPE+
info_list: TYPE = list[info]
cmd_fn:    TYPE = [info_list -> info_list]
```

```
% Resources include memory locations, processor state, and some
% devices external to the ACR.

resource: TYPE+
resource_list: TYPE = list[resource]

% Commands include processor instructions and kernel services.  Input
% resources are listed in "args", output resources in "results".  The
% function models the operation performed on the arguments.

cmd_type: TYPE = {INSTR, SEND, RCV}
command:  TYPE = [# appl_id: appl_id, cmd_type: cmd_type,
                    args: resource_list, fn: cmd_fn,
                    results: resource_list #]
cmd_list: TYPE = list[command]

% A port_id type is used to distinguish individual communication ports
% or channels implemented by the kernel.

port:  TYPE+
port:  [command -> port]
queue:    TYPE = list[info_list]

% The type res_state models the memory-like resources available to
% applications, IPC_state models the interpartition communication
% state, and system_state contains them both.

res_state:    TYPE = [resource -> info]
IPC_state:    TYPE = [port  -> queue]
system_state: TYPE = [# res: res_state, IPC: IPC_state #]

initial_res_state: res_state
initial_IPC_state: IPC_state
initial_state:     system_state = (# res := initial_res_state,
                                     IPC := initial_IPC_state #)

% Computation traces record the history of computed results

comp_event: TYPE = [# appl_id: appl_id, cmd_type: cmd_type,
                      args: resource_list, fn: cmd_fn,
                      res_res: resource_list,
                      results: info_list #]
comp_trace: TYPE = list[comp_event]


% Misc. constants

null_info: info
default_appl: appl_id
default_cmd: cmd_type
id_fn: cmd_fn = (LAMBDA (L: info_list): L)
```

```
null_comp_event: comp_event =
   (# appl_id := default_appl, cmd_type := default_cmd,
      args := null, fn := id_fn, res_res := null,
      results := null #)


%%%%%%%%%%%%%%%%

% Access control on resources omitted.


%%%%%%%%%%%%%%%%

% In the following, traces are represented in reverse chronological order,
% if read from left to right, owing to their use of the "list" data type.


% Update a list of resources within system memory with a list of values.

next_state(rlist: resource_list, values: info_list,
           s: res_state): RECURSIVE res_state =
   CASES rlist OF
     null: s,
     cons(r, rest): IF values = null
                      THEN next_state(rest, null, s) WITH [(r) := null_info]
                      ELSE next_state(rest, cdr(values), s)
                               WITH [(r) := car(values)]
                    ENDIF
   ENDCASES
   MEASURE length(rlist)


% do_step gives the values computed by a single instruction.

do_step(c: command, s: res_state): info_list =
   fn(c)(map(s)(args(c)))

% The new state that results from executing a single instruction:

execute(c: command, s: res_state): res_state =
   next_state(results(c), do_step(c, s), s)


% do_SEND, do_RCV give the values computed by a single IPC command.

do_SEND(c: command, res: res_state, IPC: IPC_state): info_list =
   map(res)(args(c))

do_RCV(c: command, res: res_state, IPC: IPC_state): info_list =
   IF IPC(port(c)) = null
      THEN null
      ELSE car(IPC(port(c)))
```

```
   ENDIF

% Execution of IPC commands is modeled by the following.

exec_SEND(c: command, res: res_state, IPC: IPC_state): system_state =
   (# res := res,
      IPC := IPC WITH [(port(c)) :=
                          append(IPC(port(c)),
                                 (: do_SEND(c, res, IPC) :) )] #)

exec_RCV(c: command, res: res_state, IPC: IPC_state): system_state =
   (# res := next_state(results(c), do_RCV(c, res, IPC), res),
      IPC := IF IPC(port(c)) = null
                THEN IPC
                ELSE IPC WITH [(port(c)) := cdr(IPC(port(c)))]
             ENDIF #)

exec_IPC(c: command, res: res_state, IPC: IPC_state): system_state =
   IF cmd_type(c) = SEND
      THEN exec_SEND(c, res, IPC)
      ELSE exec_RCV(c, res, IPC)
   ENDIF


% A system state results from the cumulative application of an entire
% command list, or from a command list segment continuing from a
% previously obtained state.

state(cmds: cmd_list): RECURSIVE system_state =
   CASES cmds OF
     null: initial_state,
     cons(c, rest): IF cmd_type(c) = INSTR
                       THEN (# res := execute(c, res(state(rest))),
                               IPC := IPC(state(rest)) #)
                       ELSE exec_IPC(c, res(state(rest)),
                                     IPC(state(rest)))
                    ENDIF
   ENDCASES
   MEASURE length(cmds)


% Construction functions for trace events

INSTR_event(c: command, res: res_state): comp_event =
   (# appl_id  := appl_id(c),
      cmd_type := cmd_type(c),
      args     := args(c),
      fn       := fn(c),
      res_res  := results(c),
      results  := do_step(c, res) #)

IPC_event(c: command, res: res_state, IPC: IPC_state): comp_event =
```

```
   (# appl_id  := appl_id(c),
      cmd_type := cmd_type(c),
      args     := args(c),
      fn       := fn(c),
      res_res  := results(c),
      results  := IF cmd_type(c) = SEND
                     THEN do_SEND(c, res, IPC)
                     ELSE do_RCV(c, res, IPC)
                  ENDIF #)

% Generate the full computation trace from the command sequence.

do_all(cmds: cmd_list): RECURSIVE comp_trace =
   CASES cmds OF
     null: null,
     cons(c, rest): cons(IF cmd_type(c) = INSTR
                           THEN INSTR_event(c, res(state(rest)))
                           ELSE IPC_event(c, res(state(rest)),
                                          IPC(state(rest)))
                        ENDIF,
                        do_all(rest))
   ENDCASES
   MEASURE length(cmds)


%%%%%%%%%%%%%%%%

% Explicit purge operations are not used.  Implicit separation of
% command streams based on IPC port is captured below.


% Computation using separated command streams involves computing both
% local and global state values.  The resource state is made global.

IPC_state_vector: TYPE = [port -> IPC_state]
trace_state_full: TYPE = [# local:  IPC_state_vector,
                            global: res_state,
                            trace:  comp_trace #]

% A command list is executed by the ensemble of partitions on one common
% processor and separate kernels for each port/communication channel.
% Each IPC command updates the kernel state for its port and the
% global resource state.

do_all_ports(cmds: cmd_list): RECURSIVE trace_state_full =
   CASES cmds OF
     null: (# local  := LAMBDA (p: port): initial_IPC_state,
              global := initial_res_state,
              trace  := null #),
     cons(c, rest):
        LET prev = do_all_ports(rest) IN
            IF cmd_type(c) = INSTR
```

```
                THEN (# local  := local(prev),
                        global := execute(c, global(prev)),
                        trace  := cons(INSTR_event(c, global(prev)),
                                       trace(prev)) #)
                ELSE (# local  :=  LAMBDA (p: port):
                                     IF p = port(c)
                                        THEN IPC(exec_IPC(c, global(prev),
                                                          local(prev)(p)))
                                        ELSE local(prev)(p)
                                     ENDIF,
                        global := res(exec_IPC(c, global(prev),
                                               local(prev)(port(c)))),
                        trace  := cons(IPC_event(c, global(prev),
                                                 local(prev)(port(c))),
                                       trace(prev)) #)
             ENDIF
   ENDCASES
   MEASURE length(cmds)


%%%%%%%%%%%%%%%%

% The proper_access predicate omitted.


%%%%%%%%%%%%%%%%

a,b:     VAR appl_id
c,d:     VAR command
cmds,c1,c2: VAR cmd_list
ct:      VAR comp_trace
init:    VAR system_state
p,q:     VAR port
r:       VAR resource
rlist:   VAR resource_list
s,s1,s2: VAR res_state
t,t1,t2: VAR IPC_state
vlist:   VAR info_list


% Lemmas expressing equality of state components under various conditions.

IPC_event_match: LEMMA
    t1(port(c)) = t2(port(c)) IMPLIES
      IPC_event(c, s, t1) =  IPC_event(c, s, t2)

res_exec_IPC_match: LEMMA
    t1(port(c)) = t2(port(c)) IMPLIES
      res(exec_IPC(c, s, t1)) =  res(exec_IPC(c, s, t2))

IPC_exec_IPC_match: LEMMA
    p = port(c) AND t1(p) = t2(p) IMPLIES
```

```
      IPC(exec_IPC(c, s, t1))(p) = IPC(exec_IPC(c, s, t2))(p)

IPC_exec_IPC_other: LEMMA
    p /= port(c) IMPLIES IPC(exec_IPC(c, s, t))(p) = t(p)


%%%%%%%%%%%%%%%%

% Following are the key lemmas relating ACR state values to those
% computed from the purged command streams.

state_invariant: THEOREM
      res(state(cmds)) = global(do_all_ports(cmds)) AND
      FORALL p: IPC(state(cmds))(p) =
                  local(do_all_ports(cmds))(p)(p)

well_partitioned: THEOREM
      do_all(cmds) = trace(do_all_ports(cmds))


END kernel_part_model
```

D.2 Proof Summary

```
 Proof summary for theory kernel_part_model
    default_appl_TCC1.....................................proved - complete
    next_state_TCC1.......................................proved - complete
    next_state_TCC2.......................................proved - complete
    next_state_TCC3.......................................proved - complete
    do_RCV_TCC1...........................................proved - complete
    state_TCC1............................................proved - complete
    state_TCC2............................................proved - complete
    do_all_TCC1...........................................proved - complete
    IPC_event_match.......................................proved - complete
    res_exec_IPC_match....................................proved - complete
    IPC_exec_IPC_match....................................proved - complete
    IPC_exec_IPC_other....................................proved - complete
    state_invariant.......................................proved - complete
    well_partitioned......................................proved - complete
    Theory totals: 14 formulas, 14 attempted, 14 succeeded.
```

D.3 Proof Chain Analysis

```
kernel_part_model.well_partitioned has been PROVED.

  The proof chain for well_partitioned is COMPLETE.

  well_partitioned depends on the following proved theorems:
    kernel_part_model.next_state_TCC2
    kernel_part_model.state_TCC2
    integers.posint_TCC1
```

```
  integers.nonneg_int_TCC1
  if_def.IF_TCC1
  kernel_part_model.state_invariant
  kernel_part_model.IPC_exec_IPC_other
  list_props.length_TCC1
  kernel_part_model.state_TCC1
  kernel_part_model.next_state_TCC3
  list_props.append_TCC1
  kernel_part_model.res_exec_IPC_match
  kernel_part_model.do_all_TCC1
  kernel_part_model.next_state_TCC1
  integers.posint_TCC2
  kernel_part_model.do_RCV_TCC1
  kernel_part_model.IPC_event_match
  kernel_part_model.IPC_exec_IPC_match

well_partitioned depends on the following axioms:
  list_adt.list_induction

well_partitioned depends on the following definitions:
  kernel_part_model.do_RCV
  kernel_part_model.state
  kernel_part_model.next_state
  list_adt.reduce_nat
  kernel_part_model.exec_RCV
  kernel_part_model.do_all_ports
  kernel_part_model.execute
  kernel_part_model.do_all
  list_props.length
  reals.>
  kernel_part_model.exec_IPC
  kernel_part_model.do_step
  list_props.append
  list_adt_map.map
  kernel_part_model.do_SEND
  reals.<=
  kernel_part_model.INSTR_event
  kernel_part_model.initial_state
  reals.>=
  kernel_part_model.exec_SEND
  notequal./=
  kernel_part_model.IPC_event
```