Presented at DCCA-7, Dependable Computing for Critical Applications, San Jose, California, USA, January 1999.

A Model of Cooperative Noninterference for Integrated Modular Avionics

Ben L. Di Vito*
NASA Langley Research Center
Hampton, VA 23681 USA
b.l.divito@larc.nasa.gov

Abstract

The aviation industry is gradually moving toward the use of integrated modular avionics (IMA) for civilian transport aircraft, potentially leading to multiple avionics functions hosted on each hardware platform. An important concern for IMA is ensuring that applications are safely partitioned so they cannot interfere with one another. On the other hand, such applications routinely cooperate, so strict separation cannot be enforced. We present a formal model for demonstrating the absence of unintentional interference in the presence of controlled information sharing among cooperating applications. The formalization draws from the techniques developed for computer security models based on noninterference concepts. Excerpts from the model formalization expressed in the language of SRI's Prototype Verification System (PVS) are included.

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 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

*This work was performed while the author was with VíGYAN, Inc., Hampton, VA 23666, USA.

ongoing standardization efforts, in particular, the work of RTCA committee SC-182, which is currently refining the ACR concept, and the recently completed ARINC 653 application executive (APEX) interface standard [1].

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. This paper sketches a formulation of partitioning requirements that has been rigorously formalized using the language of PVS (Prototype Verification System) [9]. A more detailed account of the model is available in report form [4]. This work was performed in the context of a broad program of applied formal methods activity at LaRC [2].

2 Avionics Computer Resource

The Avionics Computer Resource[1] (ACR) is an embedded generic computing platform, able to host multiple applications (avionics functions), while providing space (memory) and time (scheduling) protection. A software operating system is a fundamental part of the ACR platform, ensuring that the execution of an application does not interfere with the execution of any other application. Dedicated computer resources allocated to applications must not conflict or lead to memory, schedule, or interrupt clashes. Shared computer resources must be allocated in a way that maintains the integrity of the resources and the separation of applications.

The ACR operating system provides highly robust, kernel-level services that may be used directly by the application developer or serve as the basis for higher level services. To earn certification, kernel services must be developed in accordance with regulatory requirements such as RTCA DO-178B [10]. When applications having different levels of criticality reside on the same ACR, the kernel and other key ACR components must be qualified at or above the level of the most critical application.

Underlying all aspects of the kernel is partition management. The kernel manages partitions using a deterministic scheduling regime (e.g., 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 1 shows the ACR reference architecture envisioned by SC-182 (Level A is the most critical, Level E the least).

An ACR manages all hardware resources residing within the ACR and monitors access to all hardware resources connected to the ACR. The kernel runs on the ACR hardware with sufficient control over all hardware and software resources to ensure partitions are noninterfering. As is typically required of secure systems, this access mediation must be complete, tamper-proof, and assured.

In practice, what this means is that the kernel executes in its own protected domain with the highest privilege level available on the computer. Services are re-

[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.

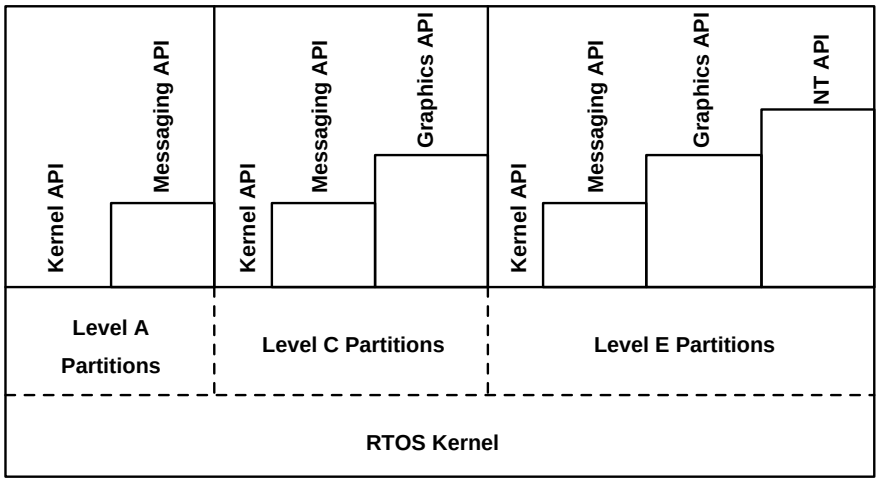

Figure 1: ACR Reference Architecture.

quested through a well-defined interface mechanism allowing users to pass parameters and receive results. Furthermore, 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 them.

3 Formalizing Partitioning

We begin the formalization discussion by motivating the approach taken. 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 paper.

3.1 Security-Oriented Noninterference

Research in computer security has been active for many years, where three broad problem areas are generally recognized: 1) confidentiality (secrecy), 2) integrity (no unauthorized modification), and 3) denial of service. Much study has been directed at defense 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 accessing a common computer system have different access permissions.

While many 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 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 [6,7] proposed the first noninterference model. Paraphrasing their model, the 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 state s those outputs viewable by v. 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.

3.2 Extensions to Noninterference

After Goguen and Meseguer's original formulation, other researchers introduced variations and extensions of their model for various purposes. Important successors were the intransitive versions of noninterference formulated by Haigh and Young [8] and Rushby [12]. Roscoe, Woodcock and Wulf introduced a noteworthy formulation based on the CSP process algebra [11], which emphasizes determinism as an overriding principle.

Recently, researchers have begun to apply noninterference concepts to model the integrity of embedded control systems. Dutertre and Stavridou developed an elegant noninterference model [5] having some similarities to our own. Their model adopts a higher level of granularity, taking task execution as the basic entity. This model also takes the important step of addressing scheduling issues to capture time partitioning properties in addition to space partitioning. What the model lacks compared to ours is a provision for cooperating partitions.

Wilding, Hardin and Greve offer another model called "invariant performance" that falls in this same line of development [13]. Although couched in somewhat different terminology, their model is likewise based on noninterference ideas. They pursue a fine-grained and concrete formalism intended to model low-level hardware mechanisms as well as kernel services. Scheduling properties are an explicit part of the model. Also included is a PVS formulation of a prototype kernel known as Schultz, along with its invariant performance properties.

While the pure 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 cooperation and communication between ACR partitions is expressly allowed, albeit under controlled conditions.

Two types of cooperation can exist in an ACR environment: direct cooperation between partitions supported by operating system services, and indirect cooperation taking place through multiple access to avionics devices. 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. Intransitive noninterference likewise could be used to capture exemptions. Instead, the modeling approach we have pursued takes the essence of these noninterference concepts and embeds them in a somewhat modified framework.

3.3 Modeling Partitioning

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." 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.

In both types of comparison, we start with identical application suites, we compare the effects of running applications in two different execution environments, then we try to rule out undesirable behaviors that might result when moving from the standard (assumed correct) architecture to the new (desired) architecture.

At the highest level, the following idealized method summarizes our approach:

- Given an ACR and its applications, map them into an equivalent federated system (each partitioned application in its own box).
- Model the externally visible behavior of the ACR with execution trace T_0. Assign traces $T_1, \ldots, T_n$ to the component behaviors in the federated system.
- 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 directly accommodated, e.g., many of those involving multiple readers and writers. By adapting the techniques of Section 5, however, these features should be within reach.

4 Noninterference Without Cooperation

As a prelude to the discussion on cooperative noninterference, we begin with the presentation of a simpler baseline model, which assumes completely separate applications. No interpartition communication (IPC) is allowed in this baseline case. Each application computes in isolation, having access only to its own resources.

4.1 Basic Framework

There are six aspects of modeling we are concerned with, each described below. This categorization is used in the full presentation [4].

- **Representation.** A minimal set of architectural concepts is provided to represent features such as a resource name space, the system resource state, and the notion of commands.
- **Computation.** Execution of command sequences and the system's response traces form the essence of computations.
- **Separation.** Consider computation using alternative command sequences, in particular, those sequences formed by purging all commands except those belonging to a single partition. Response traces resulting from the separate execution of purged command streams are compared against segments of the integrated-system trace.
- **Requirement.** Having formed 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 always holds, the same 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.** 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 Model Elements

Details of the six modeling elements are presented below.

4.2.1 Representation The collective state of all applications running on an ACR is modified in response to each instruction or kernel service. State includes main memory areas allocated to applications, register bits in the processor itself, and certain devices that have memory-like semantics. Individual state elements 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, 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 currently running application, t is the command type, a is a resource list indicating arguments to be read, f is a function with signature $f : I^* \to I^*$ representing a computation on the arguments, and r is a resource list indicating where results from function f should be written. We represent command sequences and traces by the *list* data type, semantically equivalent to that of Lisp.

4.2.2 Computation Execution of a command to produce a new value of the system state is modeled by a function $X : K \times S \to S$. The current state is defined recursively by the cumulative application of X to a command list from K^* :

$$S(\langle\,\rangle) = S_0$$
$$S(C \circ \langle\, k \,\rangle) = X(k, S(C))$$

where C is a command list, k is a command, and $\circ$ denotes the sequence or list append operation.

As computations evolve, the results produced by a command sequence 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. Construction of traces proceeds by applying the function $T : K \times S \to E$, 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^* :

$$D(\langle\,\rangle) = \langle\,\rangle$$
$$D(C \circ \langle\, k \,\rangle) = D(C) \circ \langle\, T(k, S(C)) \,\rangle$$

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.

We focus on computation traces because the domain is real time control, where it is important to ensure that outputs sent to actuators are correct. State invariants fall one step short of what is needed. It is not enough to check that memory values are appropriate; what matters is what the system *does* with such values.

4.2.3 Separation Now 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.

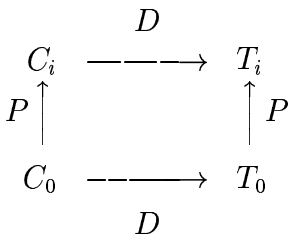

Figure 2: Trace-based partitioning requirement.

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[2] function, mapping a command list C and application ID a into the appropriate subsequence of C. We overload the purge function by adding a version of it for traces. $P : E^* \times A \rightarrow E^*$ extracts those elements of a computation trace belonging to application a.

4.2.4 Requirement 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. This formulation is similar to that of Dutertre and Stavridou [5].

Figure 2 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.

[2]The term "purge" was retained because of its historical use in noninterference models, although we now complement its selection semantics. $P(C, a)$ purges everything *not* belonging to a.

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 about the details without considering the design features themselves.

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.

4.3 PVS Formalization

A formalization of the baseline noninterference model was carried out using the language of PVS. This baseline assumes a simple IMA architecture for a system having a fixed set of applications, only a single type of command (machine instructions), and no interpartition communication. Assume further that each resource is accessible by at most one application, and resource allocation and access rights are static (permanently assigned).

Nearly all of the formalization necessary to capture the baseline model is unsurprising and we omit most of the details. We use the expressions `do_all(cmds)` to denote $D(C)$ and `purge(cmds,a)` to denote $P(C, a)$.

Let us now turn to the access control policy. Read and write access 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). 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]
allocation:   TYPE = [resource -> access_set]
```

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.

Command lists adhering to the policy must satisfy a `proper_access` predicate, which requires that for every command, the application has read access to all argument resources and write access to all result resources. A predicate `alloc` declares the access control in effect for a given system. The following condition asserts a key requirement about `alloc`, namely, that a static allocation also obeys exclusivity (at most one application has access rights to each resource).

```
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)}
```

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 nine supporting lemmas, the principal one being the following.

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

5 Cooperative Noninterference

We now consider the problem of introducing IPC services to the ACR architecture and deriving a notion of noninterference that accommodates cooperating applications. This feature is not addressed by either of the contemporary IMA models mentioned earlier [5, 13]. We draw a distinction between *resource partitioning*, protecting resources accessible to applications, and *communication partitioning*, protecting private data held by the kernel. The overall partitioning model is divided into two parts based on this distinction.

Section 5.1 describes the formalism for showing when application resources are protected from direct interference by other applications, which is an extension of the model in Section 4. Interpartition communication implemented by the kernel (or other ACR entities) presents the possibility of interference occurring within the kernel's domain. Section 5.2 develops the formalism for showing when the kernel can be considered free of flaws from this second type of interference.

We assume a basic IMA architecture having the same characteristics as before with the addition of IPC services. This leads to two classes of commands: machine instructions plus generic IPC kernel services. The exact types of communication and specific kernel services are not important for establishing resource partitioning, but they do play a role in establishing communication partitioning.

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, and we must accept the cooperative interactions while prohibiting the unintended ones.

The primary means of achieving this goal is architectural. We observe the restriction that IPC is only allowed to occur through kernel services; no shared-memory communication is permitted. Some IPC services cause updates to application-owned resources. We incorporate 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 computations of isolated processors. A global state is used to capture the computations of a part of the system we wish to hold in common for each replicated entity. The roles of local versus global will alternate for the two types of noninterference we seek to establish.

5.1 Resource Partitioning

Consider first the problem of showing that individually owned resources held by an application are shielded from direct interference. Other applications can influence resources indirectly, by sending information through IPC channels, but it should be impossible for them to access resources directly. By enforcing an access control policy on IPC services as well as processor instructions, resource partitioning can be demonstrated.

This result is obtained by replicating resource states, as before. IPC services require special treatment, however. 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 federated and integrated architectures, the exact nature of interpartition communication is immaterial.

This arrangement nevertheless complicates the elaboration of system computations. Resource states are now interdependent—it is no longer possible to separate the command sequences via $P(C, a)$ and then take the system's response to each separate stream. Doing so would miss the effects of IPC from an application's IPC partners. Hence, the elaboration of system computations is more intertwined, making concise mathematical notation difficult to achieve. Formalization using PVS functions, however, is readily accomplished. The definitions that follow show the replication of resource states while maintaining a common IPC state, as depicted in Figure 3.

We begin with the types representing the new 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.

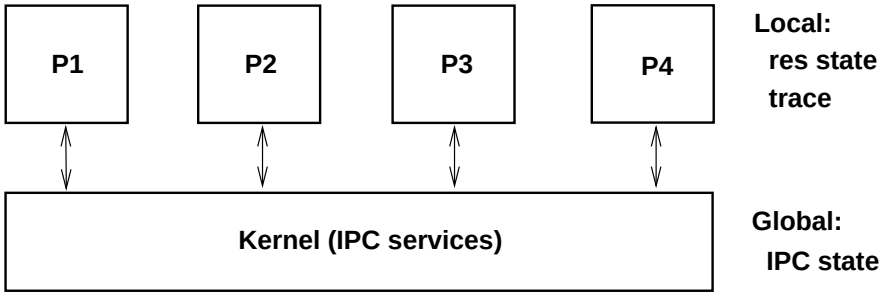

Figure 3: Global vs. local components for resource partitioning.

```
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, in the case of IPC commands, the global IPC state (Figure 4).

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.

```
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)
```

Figure 4: Computation in the resource partitioning model.

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. Consequently, an IPC command may access only those resources assigned to the partition requesting the IPC service. This is a reasonable restriction, and it is sufficient to ensure strong partitioning.

The main theorem for resource partitioning 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.

Shown below is the state invariant that holds after each command. 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))
```

5.2 Communication Partitioning

The resource partitioning requirement offers assurance against direct interference caused by other applications. 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 stated requirements.

5.2.1 Inversion of System Model Our approach is to apply the foregoing modeling framework and adapt it to the communication 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 the previous section, interchanging the roles of partitions and kernel. The partitions become the entity we hold in common while the kernel's IPC channels become the objects of separation, as if implemented by a federated system.

Application of the IPC 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 may appear less intuitive. Moreover, the approach requires modeling more of the system design than is the case with resource partitioning. It is also important to note that no guarantee of functional correctness for IPC services is inherent; the method only demonstrates that the IPC structures are independent. Nevertheless, the method offers a tractable means of addressing the question of low-level interference within an ACR's operating system.

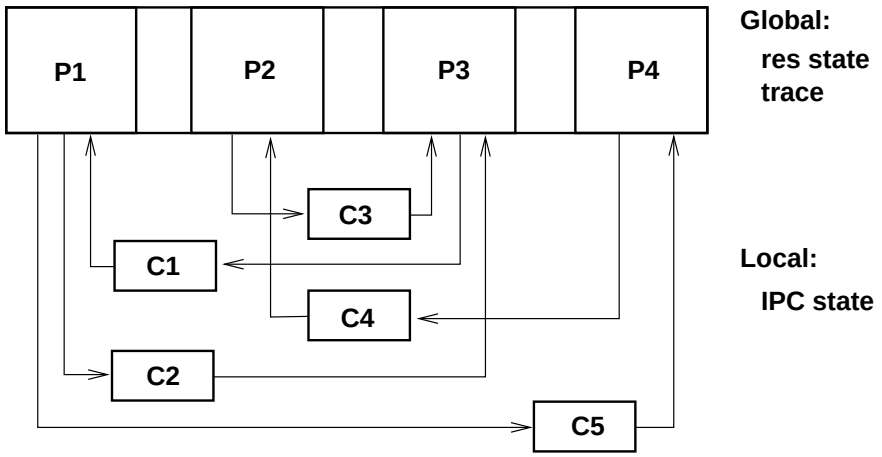

Figure 5: Global vs. local components for communication partitioning.

5.2.2 PVS Rendition Assume a port-based IPC mechanism having 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. 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 (Figure 5). 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 model of Section 5.1, 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. Figure 6 shows the details.

The function `do_all_ports` plays the same role as `do_all_purge` in the resource partitioning 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.

The main theorem for IPC partitioning 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

```
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)
```

Figure 6: Computation in the communication partitioning model.

proof was simpler than that of the previous models, owing to the simple nature of the IPC mechanism employed.

The overall state invariant 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)
```

In place of an access control policy, the communication partitioning model might require assertions about how the kernel manages its internal resources, using these assertions to establish the noninterference condition. None was used here due to a highly abstract IPC design. A more realistic design and formalization would likely require such resource management constraints.

6 Conclusion

We have presented a formal model of partitioning suitable for analyzing 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 different system architectures. 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.

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 also found to be useful in carrying out the interactive proofs, all of which were completed for the designs undertaken.

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 possible in some architectures. For this reason, a partitioning model should accommodate this type of sharing if the need arises. We have not extended our core model to cover this case, but anticipate no problems in doing so.

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 committee SC-182 has been valuable in focusing on key aspects of the avionics environment. The work of John Rushby from SRI International has likewise been useful 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.

References

[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] Ben L. Di Vito. A formal model of partitioning for integrated modular avionics. NASA Contractor Report NASA/CR-1998-208703, August 1998.

[5] Bruno Dutertre and Victoria Stavridou. A model of noninterference for integrating mixed-criticality software components. In *Proceedings of Dependable Computing for Critical Applications*, San Jose, California, January 1999.

[6] 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.

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

[8] 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.

[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. DO-178B, known in Europe as EURO-CAE ED-12B.

[11] A.W. Roscoe, J.C.P. Woodcock, and L. Wulf. Non-interference through determinism. *Journal of Computer Security*, 4(1):27–53, 1996.

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

[13] Matthew M. Wilding, David S. Hardin, and David A. Greve. Invariant performance: A statement of task isolation useful for embedded application integration. In *Proceedings of Dependable Computing for Critical Applications*, San Jose, California, January 1999.