A Model of Noninterference for Integrating Mixed-Criticality Software Components

To be presented at DCCA-7, San Jose, CA, January 99

Bruno Dutertre and Victoria Stavridou
Computer Science Laboratory, SRI International
{bruno,victoria}@csl.sri.com

Abstract

This paper examines the problem of safely integrating independent software components, of different criticality levels, in a single system. We examine the risks of interference between independent components which share common hardware resources. We propose a definition of safe integration in which only a limited form of interference is tolerated, namely a bounded performance degradation. We show how the definition can be applied to systems modeled as input-output automata, and we compare our model to other notions of non-interference and related concepts.

1 Introduction

In traditional safety-critical systems, critical functions are performed by dedicated computers running purpose-specific software. Independent functions are performed by physically separate computers and do not interfere with one another. This traditional approach is now being challenged and more integrated solutions are being proposed, such as the Integrated Modular Avionics (IMA) concept [7, 8]. In such emerging integrated systems, the hardware resource is shared between different functions and software components of mixed criticality levels may share a single processor or other hardware resource.

There is also an increased interest in developing software using off-the-shelf (OTS) components, including commercial-off-the-shelf (COTS) components and other forms of already developed software. The hope, for this approach to software development, is that developing software by integrating existing components will reduce development and maintenance costs, shorten development time, and increase productivity [3, 14]. However, the use of OTS software in high-integrity systems poses a major difficulty: key component qualities such as safety, reliability, or performance might be very difficult to establish. The internals of OTS components are rarely available for inspection, the development process might be unknown, and testing the final product is not likely to be satisfactory.

Because of these two trends, future systems may combine purpose-specific software for critical functions and OTS-based software for tasks of lower integrity. More generally, software components of different origin will coexist, performing tasks of different criticality and sharing common computing resources. Software for lower-criticality functions might not be developed to the same degree of rigor as software for critical functions, and a key issue is to ensure that the presence of lower-integrity components does not compromise the safety and reliability of the overall system. A main objective is to ensure that a piece of software whose development process might be unknown, whose performance and reliability might not be measurable, and whose design is not available for inspection will not interfere with or degrade the integrity of trusted software performing critical functions.

This paper examines the problem of ensuring that high-integrity components in an integrated system are protected from interference by other components. We assume that the high-integrity components are independent and are not intended to interact with one another or with the low-integrity components. In other words, the high-integrity components are autonomous; they perform critical functions which do not depend on any service provided by low-integrity components. In this context, the objective is to make sure that the high-integrity components behave as safely in the integrated system as they would, were they running in isolation, on separate hardware.

In Section 2, we examine the possible causes of interference between software components sharing hardware resources. We then present, in Section 3, a definition of safe integration which relates the black-box behavior of an ideal isolated component to the behavior of an integrated system which contains this component. The behavior of the two systems must be the same for all input sequences the environment may produce, modulo a limited performance degradation. Section 4 examines how safe integration can be verified when components are modeled as state machines. The integrated system can then be modeled as another state machine and we give sufficient conditions on this machine for safe integration to be achieved. Section 5 compares our noninterference model with related work on analysis of strong partitioning mechanisms for integrated systems and on extensions of security models to safety-related issues.

2 Interference between Software Components

IMA and similar systems are based on a multiprocess runtime environment. Processes which share a single processor are executing concurrently on a time-slicing basis. Process scheduling and other services are provided by a runtime environment. Communication and interaction between the processes can take place via different means which are usually implemented via shared memory. For example, the APEX interface for IMA defined in [1] supports four communication and synchronization mechanisms: semaphores, buffers, blackboards, and events, all implemented via shared memory[1].

In such systems, if two processes perform independent functions and are not intended to communicate, interference can take place through the shared resources, that is, the CPU, the physical memory, and possibly input and output devices. At least two potential sources of interference are present:

- *Data Corruption.* The nontrusted components can accidentally overwrite vital information stored in the common memory and used by the trusted components.
- *Denial of Access to Critical Resources.* The nontrusted components can prevent or delay the execution of critical functions by restricting or preventing access to a shared resource. In particular, nontrusted components can use too much CPU time or can fail to terminate or crash and hence prevent the trusted components from executing.

There is a strong analogy between these two forms of interference and two fundamental objectives of information security. Traditionally, information security aims to ensure confidentiality and integrity of sensitive data and to ensure that legitimate users are not unduly denied access to information or resources. There are clear similarities between our objectives of preventing data corruption and denial of access in a system integrating components of varying degrees of integrity and the same objectives in the security context.

The notion of secure noninterference was proposed by Goguen and Meseguer [5] to address these issues. Noninterference is the property that actions performed by a user A of a system have no effect on what another user B can observe. If user A does not interfere with user B, there is no flow of information from A to B. Similarly, we can say that a low-integrity component LO does not interfere with a high-integrity component HI if the behavior of HI is not influenced by the behavior of LO. In this sense, noninterference is an internal property of the integrated system. However, preventing internal interference may not be sufficient for safety. A nontrusted component may be able to compromise safety without internally interfering with trusted components but by accessing shared input/output channels. For example, a trusted component may be intended to control external devices such as electro-mechanical servos or actuators. In such a case, it is essential to guarantee that nontrusted components cannot have access to the same external devices. Other problems may occur if a nontrusted component is able to send messages that appear to the environment as if they originated from a trusted component or, conversely, if a nontrusted component can intercept messages destined for the trusted component.

A more general approach to ensuring safe integration is to require that the integrated system behaves like the combination of the software components running in

[1] IMA groups processes into separate units called *partitions*. Inter-process communication via the four mechanisms above is restricted to processes which belong to the same partition. Inter-partition communication relies on a different mechanism [1].

isolation. We can establish that the integrated system is safe by comparing its black-box behavior with the black-box behavior of each of the trusted components. In the following section, we give a definition of safe integration based on this approach.

3 A Model of Safe Integration

3.1 Black-box Behavior of Systems

We consider a system integrating n independent software components, sharing a single processor. Some of these components are trusted and perform a critical function. The objective is to ensure that the global system integrating all the components is safe provided the trusted components are.

In application domains such as IMA, we can assume that the trusted components must satisfy real-time constraints. We also assume that each component is deterministic. To take execution time into account, we use a discrete time model. Assume the interface of a component with the environment is characterized by an input domain I and an output domain O. An execution of the system can be described by recording at every time $t \in \mathbb{N}$ the input event of I received at that time, if any, and the output event of O produced at time t, if any. We use the special symbol $\perp$ to denote the absence of event ($\perp \notin I$ and $\perp \notin O$). An execution of the system can then be described by two sequences $\sigma = (x_t)_{t \in \mathbb{N}}$ and $\tau = (y_t)_{t \in \mathbb{N}}$, where x_t is either $\perp$ or the input event observed at time t and y_t is either $\perp$ or the output event observed at time t.

For deterministic systems, there is only one possible output sequence τ for each input sequence σ. The black-box behavior of such a system can then be represented by a mapping F from $Seq(I)$ to $Seq(O)$, where $Seq(I)$ and $Seq(O)$ denote the set of sequences of elements of $I \cup \{\perp\}$ and $O \cup \{\perp\}$, respectively[2].

An event is the observation of input values received or output values transmitted by a system at a particular time. Events need not be atomic and a single event can represent multiple communications that occur simultaneously on different ports. However, for the remainder of this paper, the exact structure of events is not relevant. The only important property is that the event space of a software component is included into the event space of the system that contains this component[3]

3.2 Safe Integration Criteria

From the environment point of view, the system integrating all the components can also be considered as a deterministic real-time system G. The input domain I_G of G includes the input domains of the components and the output domain O_G of

[2]Since past or present output cannot depend on future input events, we can assume that F satisfies the following causality property: if two input sequences $\sigma = (x_t)_{t \in \mathbb{N}}$ and $\sigma' = (x'_t)_{t \in \mathbb{N}}$ are identical up to time n (i.e. $x'_0 = x_0$, $x'_1 = x_1$, ..., $x'_n = x_n$) then the output sequences $F(\sigma)$ and $F(\sigma')$ are also identical up to time n.

[3]More rigorously, there exists a one-to-one mapping from the event space of a component to the event space of the containing system. We use inclusion for simplicity and notational convenience.

G includes the output domains of the components. The black-box behavior of G is characterized by a mapping F_G from $Seq(I_G)$ to $Seq(O_G)$. Let M be a trusted component of G with input domain $I_M \subseteq I_G$ and output domain $O_M \subseteq O_G$ and assume the behavior of M in isolation is defined by a mapping F_M. We can say that the integration of M in G is safe if, for an external observer who can only see the events in $I_M \cup O_M$, the behavior of G is identical or sufficiently close to the ideal behavior specified by F_M.

More formally, given a sequence $\sigma = (x_t)_{t\in\mathbb{N}}$ of $Seq(I_G)$, we denote by σ/I_M the sequence obtained from σ by replacing all the events not in I_M by $\perp$:

$$\sigma/I_M \;=\; (x'_t)_{t\in\mathbb{N}} \;\text{ where }\; x'_t \;=\; \begin{cases} x_t & \text{if } x_t \in I_M \\ \perp & \text{if } x_t \notin I_M. \end{cases}$$

Informally, σ/I_M is what is observed of σ when the events in $I_G - I_M$ are invisible. Similarly, for a sequence τ of $Seq(O_G)$ we denote by τ/O_M the sequence obtained by replacing all the elements of τ not in O_M by $\perp$. The behaviors of G and M are identical with respect to events in $I_M \cup O_M$ if the following condition is satisfied:

$$\forall \sigma \in Seq(I_G): \; F_G(\sigma)/O_M = F_M(\sigma/I_M).$$

In such a case, an observer who only sees events of $I_M \cup O_M$ cannot differentiate M and G: To such an observer, the two sequences σ and σ/I_M appear identical and the respective responses of G and M to σ and σ/I_M cannot be distinguished.

If the condition above is satisfied then the integration of M into G is clearly safe. However, requiring identical responses from the isolated component and the global system for all possible input sequences may be too strong a requirement in practice. The function F_M can be interpreted as the ideal behavior of component M, and in practice we can assume that G is allowed to deviate within certain limits from the ideal.

Since G performs more functions than M, a limited performance degradation, that is, a delay in producing the successive outputs, should be tolerable. We assume that a constant Δ_{out} is given which specifies how much output delay can be tolerated and we define a relation $\sqsubset$ between sequences as follows. If $\tau = (y_t)_{t\in\mathbb{N}}$ and $\tau' = (y'_t)_{t\in\mathbb{N}}$ are two sequences of elements of a set $E \cup \{\perp\}$ then $\tau \sqsubset \tau'$ if there is a bijection f from $D_\tau = \{t \mid y_t \neq \perp\}$ to the set $D_{\tau'} = \{t \mid y'_t \neq \perp\}$ such that

$$\forall t, t' \in D_\tau : t < t' \Rightarrow f(t) < f(t'),$$
$$\forall t \in D_\tau : t \leqslant f(t) \leqslant t + \Delta_{out},$$
$$\forall t \in D_\tau : y_t = y'_{f(t)}.$$

Using this notation, if $\tau = F_M(\sigma)$ is the ideal response to an input sequence σ then any τ' of $Seq(O_M)$ such that $\tau \sqsubset \tau'$ is also an acceptable response to σ. The same elements of O_M occur in the same order in τ and in τ' and each of these occurs in τ' no later than Δ_{out} time units after it occurs in τ.

To guarantee that the integration is safe, we can determine whether $F_M(\sigma/I_M) \sqsubset F_G(\sigma)/O_M$ holds for each input sequence σ the environment may produce. To satisfy this constraint, G must produce the same outputs as M and within specified deadlines. In general, it is not possible to guarantee that the deadlines are met for arbitrary input sequences. One must assume that there is a minimal delay between successive input events. Otherwise, if input events occur too frequently, G may not have enough resource to respond to them in time. We assume that a positive constant Δ_{in} gives the minimal delay between the occurrence of successive events of I_M and that another constant Δ_{in}^0 specifies the delay before the occurrence of the first event of I_M. In other words, we assume that an input sequence σ of $Seq(I_G)$ received from the environment satisfies the following constraints:

$$\forall t \in \mathbb{N} : \ \sigma_t \in I_M \Rightarrow t \geqslant \Delta_{in}^0$$
$$\forall t, t' \in \mathbb{N} : \ t < t' \wedge \sigma_t \in I_M \wedge \sigma_{t'} \in I_M \Rightarrow t' - t \geqslant \Delta_{in}.$$

We denote by Σ_{in} the set of sequences which satisfy these two conditions. The environment is assumed to produce only input sequences which belong to Σ_{in}.

Given the above assumption about the environment and the deadlines defined by the relation $\sqsubset$ we can say that the integration of M in G is safe if the following condition is satisfied:

$$\forall \sigma \in \Sigma_{in} : \ F_M(\sigma/I_M) \sqsubset F_G(\sigma)/O_M.$$

In the following section, we examine how this property can be verified when the component M and the global system G are modeled by input-output automata.

4 Proving Safe Integration

4.1 Component Model

We assume that a component running in isolation is a reactive system which is idle until input is received from the environment. Once activated by an input event, the system becomes active and starts executing and producing output events. When execution terminates, the system returns to an idle state and waits for the next input. We model such systems by a form of deterministic input-output automata which can be either in an idle state where they are waiting for input or in an active state where they are performing some computation. Any inputs submitted while the automata are in an active state are ignored. Output events are only produced during the computation steps.

This model can be used to represent systems which repeatedly perform a periodic computation, such as real-time control systems. In such cases, the input events can be assumed to be sent at regular intervals which correspond to the period of a clock. The set of input events may include a special element representing a clock tick, that carries no value and is only used to trigger a computation. This allows one to model systems that execute periodically even in the absence of input. Other systems can also be handled, where the activation events can occur at irregular intervals. For example, we can represent an emergency shutdown system by assuming that the computation is triggered when a critical event is received, without knowing when this event can occur.

More precisely, an isolated component is modeled by a state machine M of the form

$$M \quad = \quad (Q, q_0, I_M, O_M, W, \alpha, \beta, \delta)$$

where

- Q is the set of states and $q_0 \in Q$ is the initial state of M,
- I_M and O_M are the input and output spaces of M, respectively,
- W is a subset of Q,
- α is a total function from $W \times I_M$ to Q,
- β is a total function from Q to $O_M \cup \{\perp\}$,
- δ is a total function from Q to Q,

and the following condition is satisfied:

$$\forall q \in W, \;\; \delta(q) = q \;\wedge\; \beta(q) = \perp \; .$$

The set W is the set of idle states, where M is waiting for input from the environment. The functions α and β are the input and output functions, respectively, and δ is the transition function of M.

The transitions of the machine are determined by α and δ. The function α specifies how the state is updated when an input is received and δ represents a single computation step. We assume that reading an input is instantaneous and that performing a computation step takes one time unit. If M is in a state $q \in W$ at time t and input x is received at that time then M is in state $\delta(\alpha(q, x))$ at time $t + 1$. If $q \notin W$ or no input is present at time t then M goes to state $\delta(q)$ at time $t + 1$. Similarly, we assume that outputs are produced at the end of computation steps. If M enters a state q at time t and $\beta(q) \neq \perp$ then M produces the output value $\beta(q)$ at time $t + 1$. No output is ever produced in a state q of W, and M stays in such a state as long as no input event is received.

Given an input sequence $\sigma = (x_t)_{t \in \mathbb{N}}$ where $x_t \in I_M \cup \{\perp\}$, the machine M goes through a sequence of states $(q_t)_{t \in \mathbb{N}}$ where q_0 is the initial state of M and, for all $t \in \mathbb{N}$,

$$q_{t+1} \quad = \quad \begin{cases} \delta(\alpha(q_t, x_t)) & \text{if } q_t \in W \text{ and } x_t \neq \perp, \\ \delta(q_t) & \text{otherwise.} \end{cases}$$

The corresponding output sequence is $\tau = (y_t)_{t \in \mathbb{N}}$ where $y_0 = \perp$ and, for all $t \in \mathbb{N}$,

$$y_{t+1} \quad = \quad \beta(q_t).$$

The black-box behavior of M is defined by the function F_M which maps any input sequence σ to the corresponding τ.

A computation of M is started on reception of an input event x provided M is in an idle state $q \in W$. Once a computation is started, it is not interrupted by input events. The model does not guarantee that all computations eventually terminate or even that M will reach an idle state from the initial state q_0. However, if M is a trusted component we can make stronger assumptions about its behavior.

As indicated previously, a trusted component M is intended to execute in an environment where the first input event is received a delay Δ_{in}^0 after initialization and the subsequent events are separated by a delay Δ_{in} from each other. In this context, we can assume that M is ready to accept input before time Δ_{in}^0. Similarly, we can assume that any computation started in a state $q \in W$ at time t terminates before time $t + \Delta_{in}$.

Under these assumptions, there is a constant $\lambda_0 \leqslant \Delta_{in}^0$ such that

$$\begin{aligned} \delta^{\lambda_0}(q_0) &\in W \\ n < \lambda_0 \;\Rightarrow\; \delta^n(q_0) &\notin W. \end{aligned}$$

The constant λ_0 is the delay for M to initialize.

Similarly, for every state $q \in W$ and every $x \in I_M$, there is a constant $\lambda(q, x) \leqslant \Delta_{in}$ such that

$$\begin{aligned} n = \lambda(q, x) \;&\Rightarrow\; \delta^n(\alpha(q, x)) \in W \\ n < \lambda(q, x) \;&\Rightarrow\; \delta^n(\alpha(q, x)) \notin W. \end{aligned}$$

The constant $\lambda(q, x)$ is the delay for the computation triggered by x in state q to terminate.

4.2 Integrated System

Assume n independent software components are modeled by n state machines $M_1, \ldots, M_n$ as defined previously. The components share a common processor and memory space and are active alternatively under the control of a scheduling mechanism. The global system can be modeled by another state machine G whose input and output domains include the input and output domains of any machine M_i.

In every state of G, at most one of the n components is active and the behavior of G is determined by this active component. Let Q_i be the set of states of machine M_i and S be the set of states of G. A fraction of every state s of the integrated system is accessible to component M_i and this fraction corresponds to a state of Q_i. We can then assume that there is a mapping h_i from S to Q_i such that for every state $s \in S$, $h_i(s)$ is the part of s accessible by component i in the global system. If component i is active in a state s, then the behavior of G in s is the same as the behavior of M_i in $h_i(s)$: the same output is produced and the next global state s' is such that $h_i(s') = \delta_i(h_i(s))$, where δ_i is the transition function of M_i.

With respect to input events, we can no longer assume that G accepts input only when it is idle, that is, when none of the components is active. To ensure that deadlines are met, G must have the capability to interrupt the active component when urgent events are received. Also, an input received at time t may not be destined for the component active at that time. G must be able to accept such an input on behalf of a nonactive component.

As a whole, we model the integration of n components $M_1, \ldots, M_n$, where

$$M_i = (Q_i, q_0^i, I_{M_i}, O_{M_i}, W_i, \alpha_i, \beta_i, \delta_i),$$

as follows. The global system is represented by an automaton G of the form

$$G = (S, s_0, I_G, O_G, \alpha, \beta, \delta),$$

where

- S is the set of states and $s_0 \in S$ is the initial state of G,
- I_G and O_G are the input and output spaces of G,
- α is a total function from $S \times (I_G \cup \{\perp\})$ to S,
- β is a total function from S to $O_G \cup \{\perp\}$,
- δ is a function from S to S,

and the following condition is satisfied:

$$\forall s \in S : \ \alpha(s, \perp) = s.$$

As previously, α is the input function, β the output function, and δ the transition function of G. If G is in state s at time t and an input event $x \in I_G$ is received then G transitions to $\delta(\alpha(s, x))$ in one time unit. If no input is present then G only performs the computation step and enters state $\delta(\alpha(s, \perp)) = \delta(s)$ at time $t + 1$. Output values are also produced in the same way as previously. The only difference with the machine model used for isolated components is that input may be accepted at any time.

Given an input sequence $\sigma = (x_t)_{t \in \mathbb{N}}$, where $x_t \in I_G \cup \{\perp\}$ then G goes through the sequence of states $(s_t)_{t \in \mathbb{N}}$ where s_0 is the initial state of G and

$$s_{t+1} = \begin{cases} \delta(\alpha(s_t, x_t)) & \text{if } x_t \neq \perp, \\ \delta(s_t) & \text{otherwise.} \end{cases}$$

and produces an output sequence $\tau = (y_t)_{t \in \mathbb{N}}$ defined by $y_0 = \perp$ and $y_{t+1} = \beta(s_t)$. G is deterministic. For every σ, G produces a unique output sequence $\tau = F_G(\sigma)$ defined as above.

The global system G and the components $M_1, \ldots, M_n$ are related in the following way. For every component M_i, I_{M_i} is a subset of I_G and O_{M_i} a subset of O_G, and there exist a mapping h_i from S to Q_i and a subset A_i of S such that the three following conditions are satisfied:

$$\forall s \in A_i : \; h_i(\delta(s)) = \delta_i(h_i(s)) \;\wedge\; \beta(s) = \beta_i(h_i(s)),$$
$$\forall s \in S, x \in I_{M_i} : \; h_i(s) \in W_i \Rightarrow h_i(\alpha(s,x)) = \alpha_i(h_i(s), x),$$
$$h_i(s_0) \; = \; q_0^i.$$

Furthermore, the sets $A_1, \ldots, A_n$ are mutually disjoint.

The set A_i represents the set of states of G where component M_i is active. The first condition above specifies that the output of G in such a state s is the same as the output of M_i in $h_i(s)$ and that the part of the state visible to M_i is updated in G in the same way as it would be in M_i. The second condition means that an input event x relevant to M_i and received in a state s where M_i is ready for input has the same effect on $h_i(s)$ as if received by M_i.

One can think of α as the handling of interrupts by a runtime executive. If an input x is received in state s, then the current computation is interrupted, the runtime executive updates the state to $\alpha(s,x)$, and execution proceeds from there. If all interrupts are masked in a state s then we set $\alpha(s,x) = s$ for all x. States where some interrupts are masked and others are accepted can be modeled in a similar way. Also the union of the sets $A_1, \ldots, A_n$ is not necessarily equal to S; there may be states s where no component is active. This allows us to model computation steps performed by the runtime executive, such as switching execution contexts.

The transition function δ models atomic computation steps that are not interruptible and take one time unit to execute. A computation steps performed when component M_i is active may have side effects not directly visible to M_i: the function δ may modify elements of the state s which are not accessible to M_i. This provides extra generality and may be useful to model the behavior of the runtime executive. For example, the kernel may maintain a state variable to hold the active process, and the active component in state $\delta(s)$ may be different from the active component in s, that is, M_i.

4.3 Sufficient Conditions for Safe Integration

In this section, a fixed component M_i of G is assumed to be trusted. Three constants Δ_{out}, Δ_{in}^0, and Δ_{in} which characterize the task performed by M_i are given. The objective is to ensure that the integration of M_i in G is safe, according to the criterion defined in Section 3.2:

$$\forall \sigma \in \Sigma_{in} : \; F_{M_i}(\sigma / I_{M_i}) \sqsubset F_G(\sigma)/O_{M_i}.$$

We give several conditions on G which are sufficient to ensure that this property is satisfied. These conditions can be classified as noninterference constraints and scheduling constraints. The noninterference conditions ensure that the activity of components other than M_i does not have a visible effect on the global state as perceived by M_i. The scheduling conditions ensure that M_i is allocated sufficient CPU time so that the required deadlines are met.

The noninterference conditions are the following:

$$\forall s \in S : \; s \notin A_i \;\Rightarrow\; h_i(\delta(s)) = h_i(s),$$
$$\forall s \in S : \; s \notin A_i \;\Rightarrow\; \beta(s) \notin O_{M_i},$$
$$\forall s \in S, x \in I_G : \; x \notin I_{M_i} \;\Rightarrow\; h_i(\alpha(s,x)) = h_i(s).$$

The first constraint requires that computations performed by the runtime executive or by components other than M_i have no effect on the part of the state visible to M_i. The second constraint requires that no component other than M_i can access M_i's output interface. The last constraint requires that input events not destined for M_i have no effect on what M_i can perceive of the global state.

To specify the scheduling constraints, we need to measure the amount of CPU time allocated to M_i after a finite number of computation steps of G. Given u a finite sequence of elements of $I_G \cup \{\bot\}$ and s a state of G, we denote by $\hat{\delta}(s,u)$ the state reached by G starting from s and after the sequence u of input events has been received. The function $\hat{\delta}$ can be defined as follows:

$$\begin{aligned} \hat{\delta}(s,\epsilon) &= s \\ \hat{\delta}(s,u.x) &= \delta(\alpha(\hat{\delta}(s,u),x)), \end{aligned}$$

where ϵ denotes the empty sequence, $x \in I_G \cup \{\bot\}$, and $u.x$ is the concatenation of u and x. The amount of CPU time allocated to M_i during the finite computation defined by s and u is denoted by $\mu(s,u)$ and it is the number of states of A_i visited between s and $\hat{\delta}(s,u)$:

$$\begin{aligned} \mu(s,\epsilon) &= 0 \\ \mu(s,u.x) &= \begin{cases} 1 + \mu(s,u) & \text{if } \alpha(\hat{\delta}(s,u),x) \in A_i, \\ \mu(s,u) & \text{otherwise.} \end{cases} \end{aligned}$$

Let B be the set of finite input sequences of $I_G \cup \{\bot\}$ which do not contain events of I_{M_i}. We denote by $\mu(s,m)$ the minimal amount of CPU allocated to M_i for all finite executions of length m, started in s, and during which no event of I_M is received:

$$\mu(s,m) \;=\; \min\{\, \mu(s,u) \mid u \in B \wedge |u| = m \,\}.$$

The scheduling constraints can be organized in two sets. The two constraints below ensure that G meets the deadlines during the initialization phase:

$$\mu(s_0, \Delta^0_{in}) \geqslant \lambda_0$$
$$\forall m \in \mathbb{N} : \; \delta_i^m(q_0) \in Out_i \;\Rightarrow\; \mu(s_0, m + \Delta_{out}) \geqslant m,$$

where Out_i is the set of states q of M_i such that $\beta(q) \neq \perp$ and λ_0 is the time for M_i to terminate the initial computation (cf. Sect. 3.2).

Given any global state s such that $h_i(s) \in W_i$ and any event $x \in I_{M_i}$, the second set of constraints is

$$\mu(\alpha(s, x), \Delta_{in}) \geqslant \lambda(h_i(s), x),$$
$$\forall m \in \mathbb{N}: \ \delta_i^m(\alpha_i(h_i(s), x)) \in Out_i \ \Rightarrow \ \mu(\alpha(s, x), m + \Delta_{out}) \geqslant m.$$

The constant $\lambda(h_i(s), x)$ is the delay for M_i to terminate the computation started in $h_i(s)$ on reception of event x.

In conjunction with the noninterference conditions, these four conditions ensure that the deadlines are met. Two constraints require that the amount of CPU time allocated to component M_i is sufficient for M_i to terminate before the deadline Δ_{in}^0 or Δ_{in}. The two other constraints are similar and ensure that the output deadlines are met: if M_i isolated produces an output after delay m then G will produce the same output before a delay $m + \Delta_{out}$.

It can be shown that if all the noninterference and scheduling constraints are satisfied then $F_{M_i}(\sigma/I_{M_i}) \sqsubset F_G(\sigma)/O_{M_i}$ for all input sequence σ of Σ_{in}. The proof relies on the following lemmas:

Lemma 1 *If the noninterference conditions are satisfied then for any global state s and any sequence $u \in B$, if $\mu(s, u) = m$ then*

$$h_i(\hat{\delta}(s, u)) \ = \ \delta_i^m(h_i(s)).$$

Lemma 2 *If the noninterference conditions are satisfied, s is a state of G, and u is a sequence of B then for any $m < \mu(s, u)$, there is a unique prefix $v.x$ of u such that $\alpha(\hat{\delta}(s, v), x) \in A_i$ and $\mu(s, v) = m$.*

The noninterference and scheduling constraints given above ensure that the integration of component M_i in G is safe. The model is independent of any particular mechanism for implementing these constraints. The noninterference conditions characterize the impossibility of a process to corrupt data owned by another process. Preventing such data corruption requires *space partitioning*. Typically space partitioning is achieve using memory protection mechanisms provided by hardware. If the scheduling conditions are satisfied, process M_i is guaranteed access to a sufficient amount of CPU time. Ensuring the scheduling conditions requires *temporal partitioning* that can be implemented using timer interrupts to stop and take control away from overrunning processes. The various architectural and implementation issues pertaining to both spatial and temporal partitioning are discussed at large in [11].

5 Discussion and Related Work

In critical applications, the integration of multiple functions on shared hardware requires partitioning mechanisms that provide, with high assurance, a similar level of fault containment as the federated architectures of today. The integration model presented in this paper define properties these partitioning mechanisms must achieve to ensure that non-trusted software components do not interfere with a trusted one, in a system where no intercomponent communication is intended.

As mentioned previously, there are clear similarities between safe integration and aspects of computer security and our approach is inspired from the security notions of noninterference [5, 6] and separability [10, 12]. These two concepts give criteria to guarantee that what one user does has no influence on what another user can observe. In the context of safe integration, we are more concerned with what components can do than with what they can observe and it is easy to adapt noninterference and separability to this purpose. However, in their classic forms, noninterference and separability do not address timing issues which are crucial in integrated real-time applications. Another limit of security models in our context is that they usually intend to prevent all interference while in practice some limited and controlled interference is necessary or unavoidable. The main contribution of our model is the definition of safe integration conditions involving both timing and spatial noninterference and where bounded timing interference is tolerated.

Some of the assumptions made in Sect. 3 on the behavior of software components could be relaxed. For example, it is not difficult to extend the model to include nondeterministic components. The automata-based model developed in Sect. 4 also makes particular assumptions about task execution and access to input and output. Since the possible scheduling and communication mechanisms involved can vary widely from one platform to the next, it is difficult to develop a model general enough to apply in all contexts. The details can be adjusted. The important element of the approach is the method to establish that safe integration is achieved, based on the use of mappings between global and component machines.

A more serious limitation is that our model only applies to systems where components do not communicate. In practice, most integrated systems require some form of cooperation between the different processes or tasks. We are working on extending the approach to this general case. An important issue in this more general context is ensure that there is no source of interference between processes other than via explicitly authorized communication channels. Di Vito [4] develops several partitioning models for IMA using PVS [9] where communication issues are addressed. The approach is closely related to the security notion of noninterference developed by Goguen and Meseguer [5]. Timing is ignored and only spatial partitioning is considered. In the case of cooperating processes, the partitioning requirements are based on comparing the behavior of the integrated system to the behavior of an equivalent federated system (see [4] for details). Other models are also examined with slightly different partitioning requirements. In particular a technique is developed to show that the kernel properly isolates the communication channels from one another.

Like ours, Di Vito's model is based on adapting the security concepts of non-interference and separation to integrated systems. Wilding et al. [15] propose a different approach to verifying partitioning in integrated systems. Each partition is allocated a fixed memory segment and is allocated a periodic window of CPU time on a cyclic basis. The verification of partitioning is based on comparing two systems configured identically with respect to a given partition P. The same memory segment and is allocated to P and the same global schedule is used in both systems. The partitioning mechanisms are correct if the content of the memory segment of P is the same at all time in the two systems, provided it is the same initially. This criterion addresses both timing and spatial partitioning but does not allow cooperation between partitions. The correctness criterion is specific of the detailed hardware/software model used and in particular of the adoption of a fixed scheduling. However, adapting the criterion to other scheduling policies does not seem to pose insurmountable difficulties.

Outside of the IMA context, Totel et al. [13] apply security-related notions to dependable system integration. They define an access control model for supporting object-oriented software components of multiple integrity levels in a single systems. The model includes ordinary object of a single integrity level and objects whose integrity level varies during execution. The access control policy extends Biba's model of data-integrity for multilevel security [2] and intends to ensure that data does not flow from objects of low integrity levels to object of higher integrity levels. The only exception to this rule allows a special class of so-called validation objects to upgrade data from low to high integrity levels. This model gives an approach to defining how software components of different criticality levels may cooperate safely.

The models discussed above address various aspects of a general dependable integration problem. For integrated architectures such as IMA to be acceptable in safety-critical applications one must develop mechanisms to prevent unintended interaction between components and provide high assurance that these mechanisms are adequate. As discussed in [11], there is a large space of architectural and design choices that can achieve the required partitioning. The challenge is to define models that are general enough to encompass spatial, temporal, and communication aspects. The models above address memory partitioning and to some extent temporal partitioning and intercomponent communication in the context of single-processor systems. A lot remains to be done in the areas of temporal partitioning and communication. Subtle interactions such as the impact of interrupts on timing and temporal partitioning are not yet adequately treated. It also remains to investigate the integration and partitioning issues posed by distributed systems. New risks of interference arise in such systems due to shared communication media such as avionics buses. New models need to be developed to analyze bus access conflicts and bus scheduling issues, and the impact of bus communication on the local memory and temporal partitioning defined for a single-processor hardware component of the system.

6 Conclusion

This paper proposes a model of safe integration allowing one to mix software components of different criticality levels and performing independent functions in a single system. The main principle is to ensure that the black-box behavior of the resulting integrated system does not deviate too much from the behavior of each of the high-integrity components. Based on this definition, one can derive sufficient conditions of safe integration which rely on an automata model.

The notion of safe integration and the model used are related to the two security concepts of separability and noninterference. The main differences are due to two crucial objectives: maintaining availability and responsiveness of trusted components in the integrated system. Timing issues become essential.

In future work, we envisage relating safe integration and software architecture. The main idea would be to develop software architectures for integrated systems by successive refinements, starting from a top-level description where components are isolated and by applying transformations ensuring that safe integration criteria as defined in this paper are satisfied. This approach could then be extended to cooperating components. The successive refinements would then have to ensure that no risk of interference is introduced other than via the communication channels explicitly present in the top-level description.

References

[1] Avionics Application Software Standard Interface. ARINC Specification 653, ARINC, Annapolis, Maryland, January 1997.

[2] R.R. Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3135, Mitre Corporation, April 1977.

[3] A. Brown, D.J. Carney, and M.D. McFalls. Proceedings of the SEI/MCC Symposium on the Use of COTS in Systems Integration. Technical Report CMU/SEI-95-SR-007, Software Engineering Insitute, June 1995.

[4] Ben Di Vito. A Model of Cooperative Noninterference for Integrated Modular Avionics. In *Dependable Computing for Critical Applications (DCCA-7)*, San Jose, CA, January 1999. This volume.

[5] J.A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the Symposium on Security and Privacy*, pages 11–20, Oakland, CA, 1982.

[6] J.A. Goguen and J. Meseguer. Unwinding and Inference Control. In *Proceedings of the Symposium on Security and Privacy*, pages 75–86, Oakland, CA, 1984.

[7] M. Morgan. Integrated Modular Avionics for Next-Generation Commercial Airplanes. *IEEE Aerospace and Electronic Systems*, 6(8):9–12, August 1991.

[8] A. Nadesakumar, R. Crowder, and C. Harris. Advanced system concepts for future civil aircraft - an overview of avionic architecture. *Proceedings of the Institution of Mechanical Engineers. Part G: Journal of Aerospace Engineering*, 209(G4), 1995.

[9] S. Owre, J. Rushby, N. Shankar, and F. 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] J. Rushby. Design and Verification of Secure Systems. In *Proceedings of the 8th Symposium on Operating Systems Principles, ACM SIGOPS*, volume 26, pages 54–60, November 1981.

[11] J. Rushby. Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance. Draft technical report, Computer Science Laboratory, SRI International, October 1998.

[12] J. Rushby and B. Randell. A Distributed Secure System. *IEEE Computer*, 16(7):55–67, July 1983.

[13] E. Totel, J.-P. Blanquart, Y. Deswarte, and D. Powell. Supporting Multiple Levels of Criticality. In *FTCS-28, The 28th Annual Fault Tolerant Computing Symposium*, pages 70–79, Munich, Germany, June 1998.

[14] M.R Vigder, W.M. Gentleman, and J. Dean. COTS Software Integration: State of the Art. Technical Report 39198, National Research Council of Canada, January 1996.

[15] M. Wilding, D. Hardin, and D. Greve. Invariant Performance: A Statement of Task Isolation Useful for Embedded Application Integration. In *Dependable Computing for Critical Applications, DCCA-7*, San Jose, CA, January 1999. This volume.