GUARDS: A Generic Upgradable Architecture for Real-Time Dependable Systems

D. Powell, *Member, IEEE,* J. Arlat, *Member, IEEE,* L. Beus-Dukic,
A. Bondavalli, *Member, IEEE Computer Society,* P. Coppola, A. Fantechi,
E. Jenn, C. Rabéjac, and A. Wellings

Abstract—The development and validation of fault-tolerant computers for critical real-time applications are currently both costly and time consuming. Often, the underlying technology is out-of-date by the time the computers are ready for deployment. Obsolescence can become a chronic problem when the systems in which they are embedded have lifetimes of several decades. This paper gives an overview of the work carried out in a project that is tackling the issues of cost and rapid obsolescence by defining a generic fault-tolerant computer architecture based essentially on commercial off-the-shelf (COTS) components (both processor hardware boards and real-time operating systems). The architecture uses a limited number of specific, but generic, hardware and software components to implement an architecture that can be configured along three dimensions: redundant channels, redundant lanes, and integrity levels. The two dimensions of physical redundancy allow the definition of a wide variety of instances with different fault tolerance strategies. The integrity level dimension allows application components of different levels of criticality to coexist in the same instance. The paper describes the main concepts of the architecture, the supporting environments for development and validation, and the prototypes currently being implemented.

Index Terms—Computer architecture, generic architecture, embedded systems, fault tolerance, real-time, integrity levels.

1 Introduction

Most ultradependable real-time computing architectures developed in the past have been specialized to meet the particular requirements of the application domain for which they were targeted. This specialization has led to very costly, inflexible, and often hardware-intensive solutions that, by the time they are developed, validated, and certified for use in the field, can already be out-of-date in terms of their underlying hardware and software technology. This problem is exacerbated in some application domains since the systems in which the real-time architecture is embedded may be deployed for several decades, i.e., almost an order of magnitude longer than the typical lifetime of a generation of computing technology.

A consortium of European companies and academic partners has been formed to design and develop a Generic Upgradable Architecture for Real-time Dependable Systems (GUARDS), together with an associated development and validation environment. The end-user companies in the consortium all currently deploy ultradependable real-time embedded computers in their systems, but with very different requirements and constraints resulting from the diversity of their application domains: nuclear submarine, railway, and space systems. The overall aim of the GUARDS project is to significantly decrease the lifecycle costs of such embedded systems. The intent is to be able to configure *instances* of a generic architecture that can be shown to meet the very diverse requirements of these (and other) critical real-time application domains. A three-pronged approach is being followed to reduce the cost of validation and certification of instances of the architecture: 1) design for validation, so as to focus validation obligations on a minimum set of critical components; 2) reuse of already-validated components in different instances; and 3) the support of software components of different criticalities.

The paper is structured as follows: Section 2 sketches the rationale for the design of the generic architecture, which is then summarized in Section 3. Central to the architecture is an interchannel communication network, which is described in Section 4. Section 5 details the interchannel fault tolerance mechanisms while Section 6 discusses the scheduling issues raised by active replication of real-time tasks. Sections 7 and 8 discuss, respectively, the development and validation environments that accompany the architecture. Section 9 describes the prototypes currently being implemented. Finally, Section 10 concludes the paper.

2 Design Rationale

To merit the epithet "generic," the architecture must be able to meet the widest possible spectrum of dependability and real-time requirements. To this end, we first consider some key nonfunctional requirements of typical applications in

- *D. Powell and J. Arlat are with LAAS-CNRS, 7 avenue du Colonel Roche, Toulouse Cedex 4, France. E-mail: david.powell@laas.fr.*
- *L. Beus-Dukic and A. Wellings are with the University of York, York YO10 5DD, United Kingdom.*
- *A. Bondavalli is with PDCC-CNUCE-CNR, Via S. Maria 36, 56126 Pisa, Italy.*
- *P. Coppola is with Intecs Sistemi S.p.A., V.L. Gereschi 32/34, 56127 Pisa, Italy.*
- *A. Fantechi is with PDCC-IEI CNR, Via S. Maria 46, 56126 Pisa, Italy.*
- *E. Jenn is with Technicatome, BP 34000, 13791 Aix-en-Provence Cedex 3, France.*
- *C. Rabéjac is with Matra Marconi Space, 31 rue des Cosmonautes, 31402 Toulouse Cedex 4, France.*

For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 109045.

each of the three end-user domains. We then discuss the issues of fault classes and real-time scheduling.

2.1 Key Nonfunctional Requirements

A typical instance of the architecture in the railway domain would be a fail-safe control system. Standards in this domain dictate extremely low catastrophic failure rates for individual subsystems (e.g., less than 10^{-11}/hour with respect to physical faults). In railway applications, it is common to physically segregate subsystems responsible for vital (safety-critical) functions from nonvital functions. We decided to investigate the possibility of a single instance supporting both high-integrity vital functions and low-integrity nonvital functions.

In the nuclear submarine domain, an instance of the architecture would typically be used to support secondary protection functions, which are required to be ready to react in case of (rare) incidents. Two requirements from this application domain impose quite severe restrictions on the design space. First, it must be possible to separate redundant elements of the architecture by several meters so as to tolerate physical damage. Second, to avoid obsolescence during the submarine's lifetime, the use of unmodified commercial off-the-shelf operating system(s) is mandatory.

A particularly challenging application in the space domain is that of an autonomous spacecraft carrying out missions containing phases that are so critical that tolerance of several faults may be required (e.g., target fly-by or docking). During noncritical phases, the redundant elements may be powered down to save energy. Moreover, it is necessary for an instance to be able to support software of different integrity levels: high-integrity critical software that is essential for long-term mission reliability and potentially unreliable payload software.

2.2 Fault Classes

The architecture aims to tolerate permanent and temporary physical faults (of both internal and external origins) and should provide tolerance or confinement of software design faults. This wide spectrum of fault classes [42] has several consequences beyond the basic physical redundancy necessary to tolerate permanent internal physical faults. Tolerance of permanent external physical faults (e.g., physical damage) requires geographical separation of redundancy. Temporary external physical faults (transients) can lead to rapid redundancy attrition unless their effects can be undone. This means that it must be possible to recover corrupted processors. Temporary internal physical faults (intermittents) are treated as either permanent or transient faults according to their rate of recurrence.

Many design faults can also be tolerated like intermittents if their activation conditions are sufficiently diversified [29] (e.g., through loosely coupled replicated computations). However, design faults that are activated systematically for a given sequence of application inputs can only be tolerated through diversification of design or specification. Due to limited resources, the project has not considered diversification of application software beyond imposing the requirement that no design decision should preclude that option in the future. However, we have studied the use of integrity level and control-flow monitoring mechanisms to ensure that design faults in noncritical application software do not affect critical applications. Moreover, we have considered diversification for tolerating design faults in off-the-shelf operating systems. We also encourage activation condition diversification to provide some tolerance of design faults in replicated hardware and replicated applications.

2.3 Real-Time Models

In keeping with the genericity objective, the architecture must be capable of supporting a range of real-time computational and scheduling models.

The *computational model* defines the form of concurrency (e.g., tasks, threads, asynchronous communication, etc.) and any restriction that must be placed on application programs to facilitate their timing analysis (e.g., bounded recursion). Applications supported by GUARDS may conform to a time-triggered, event-triggered or mixed computational model.

Three *scheduling models* are considered [69]:

- Cyclic—as typified by the traditional cyclic executive.
- Cooperative—where an application-defined scheduler and the prioritized application tasks explicitly pass control between one another to perform the required dispatching.
- Preemptive—the standard preemptive priority scheme.

We have focused primarily on the preemptive scheduling model since this is the most flexible and the one that presents the greatest challenges.

3 The Generic Architecture

The diversity of end-user requirements and fault tolerance strategies led us to define a generic architecture that can be configured into a wide variety of instances. The architecture favors the use of commercial off-the-shelf (COTS) hardware and software components, with application-transparent fault tolerance implemented primarily by software. Drawing on experience from systems such as SIFT [47], MAFT [37], FTPP [30], and Delta-4 [52], the generic architecture is defined along three dimensions of fault containment (Fig. 1) [53]:

- Integrity levels, or design-fault containment regions.
- Lanes, or secondary physical-fault containment regions.
- Channels, or primary physical-fault containment regions.

A particular instance of the architecture is defined by the dimensional parameters $\{C, M, I\}$, a reconfiguration strategy, and an appropriate selection of generic hardware and software GUARDS components. These generic components implement mechanisms for:

- Interchannel communication.
- Output data consolidation.
- Fault tolerance and integrity management.

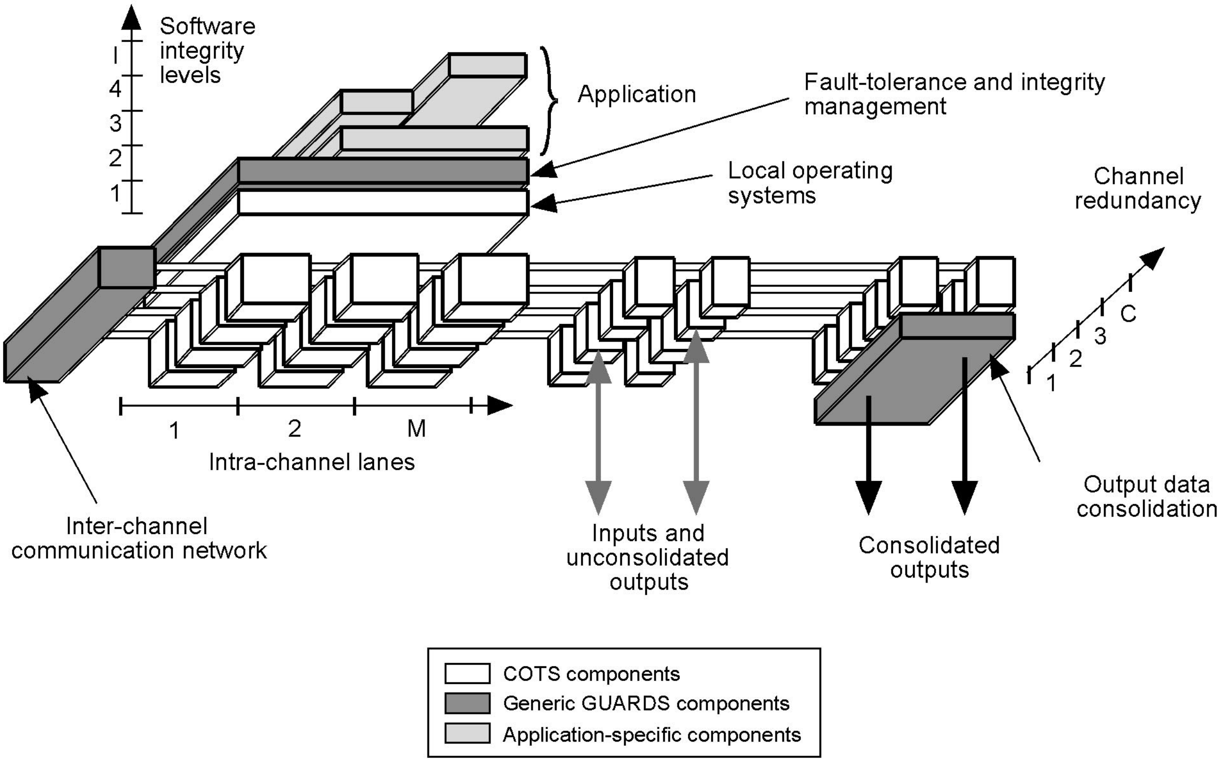

Software integrity levels
I
4
3
2
1
Application
Fault-tolerance and integrity management
Local operating systems
Channel redundancy
C
3
2
1
1
2
M
Intra-channel lanes
Inter-channel communication network
Inputs and unconsolidated outputs
Consolidated outputs
Output data consolidation
□ COTS components
▨ Generic GUARDS components
░ Application-specific components

Fig. 1. The generic architecture.

Fault tolerance and integrity management are software-implemented through a distributed set of generic system components (shown as a "middleware" layer on Fig. 1). This layer is itself fault-tolerant (through replication and distribution of its components) with respect to faults that affect channels independently (e.g., physical faults). However, the tolerance of design faults in this system layer is not explicitly addressed.[1]

3.1 The Integrity Dimension

The integrity dimension aims to provide containment regions with respect to software design faults. The intent is to protect critical components from the propagation of errors due to residual design faults in less-critical components. Each application object is classified within a particular integrity level according to how much it can be trusted (the more trustworthy an object is, the higher its integrity level). The degree to which an object can be trusted depends on the evidence that is available supporting its correctness, and the consequences of its failure (i.e., its criticality).

The required protection is achieved by enforcing an integrity policy to mediate the communication between objects of different levels. Basically, the integrity policy seeks to prohibit flows of information from low to high integrity levels, like in the Biba policy [15]. However, this approach is inflexible. An object can obtain data of higher integrity than itself, but the data must then inherit the level of integrity of this object. This results in a decrease in the integrity of the data, without any possibility of restoring it. We deal with this drawback by providing special objects (*Validation Objects*) whose role is to apply fault tolerance mechanisms on information flows. The purpose of these objects is to output reliable information by using possibly corrupted data as input (i.e., with a low integrity level). Such objects upgrade the trustworthiness of data and, hence, allow information flows from low to high integrity levels [67].

It must be ensured that it is not possible to by-pass the controls put into place to enforce the policy. This is achieved by spatial and temporal isolation, which are provided, respectively, by memory management hardware and resource utilization budget timers [66]. Furthermore, for the most critical components (the topmost integrity level) and a core set of basic components (i.e., the integrity management components and the underlying hardware and operating systems), it must be assumed either that there are no design faults or that they can be tolerated by some other means (e.g., through diversification).

In this paper, we do not detail the integrity dimension any further—the interested reader should refer to [66], [67].

3.2 The Lane Dimension

Multiple processors or *lanes* are used essentially to define secondary physical fault containment regions. Such secondary regions can be used to improve the capabilities for fault diagnosis within a channel, e.g., by comparison of computation replicated on several nodes. There is also scope for improving coverage with respect to design faults by using intrachannel diversification.

[1] Note, however, that correlated faults are included in the models used to assess the dependability of instances of the architecture: See Section 8.2.

Alternatively, lanes can be used to improve the availability of a channel, e.g., by passivating a node that is diagnosed to be permanently faulty. The required fault diagnosis could be triggered either by the error-processing mechanisms within a channel or through an error signal from the interchannel voting mechanisms.

Further reasons for defining an instance with multiple lanes include parallel processing to improve performance and isolation of software of different integrity levels. To aid the timing analysis of such software we require that the multiple processors within a channel have access to shared memory (see Section 6).

3.3 The Channel Dimension

Channels provide the *primary* fault containment regions that are the ultimate line of defense within a single instance for physical faults that affect a single channel. Fault tolerance is based on active replication of application tasks over the set of channels. It must be ensured that replicas are supplied with the same inputs in the same order, despite the occurrence of faults. Then, as long as replicas on fault-free channels behave deterministically, they should produce the same outputs. Error processing can thus be based on comparison or voting of replica outputs.

Not all instances require the same number of channels. In fact, one could imagine an instance with just one channel. This would be the case for an application that only requires multiple integrity levels or for which the fault tolerance mechanisms implemented within a channel are judged to be sufficient. It should be expected, however, that most applications require instances with several channels. Important cases are:

- Two channels: motivated either by a requirement for improved safety (using interchannel comparison) or improved reliability (based on intrachannel self-checking to provide crash failure semantics).
- Three channels: the well-known triple modular redundancy (TMR) strategy that enables most[2] faults in one channel to be masked. In addition, any disagreements are detected and used as inputs for error diagnosis and fault treatment.
- Four channels: to enable masking of completely arbitrary faults or to allow a channel to be isolated for off-line testing while still guaranteeing TMR operation with the remaining on-line channels.

Instances of the architecture with more than four channels are not currently envisaged.

4 Interchannel Communication Network

Central to the architecture is an interchannel communication network (ICN), which fulfills two essential functions:

- It provides a global clock to all channels.
- It allows channels to achieve interactive consistency (consensus) on nonreplicated data.

The ICN consists of an ICN-manager for each channel and unidirectional serial links to interconnect the ICN-managers. In the current implementation, the ICN-manager is a Motorola 68040-based board with a dual-port shared memory for asynchronous communication with the intra-channel VME back-plane bus. Serial links are provided by two Motorola 68360-based piggyback boards. Each such board provides two Ethernet links. One link is configured as transmit only, the other links are configured as receive only. An ICN-manager can thus simultaneously broadcast data to the remote ICN-managers over its outgoing serial link and receive data from the remote ICN-managers over the other links.

4.1 Clock Synchronization

The ICN-managers constitute a set of fully interconnected nodes. Each node has a physical clock and computes a global logical clock time through a fault-tolerant synchronization algorithm. Such an algorithm is classically defined as one that satisfies both the agreement and accuracy properties:

- The *agreement* condition is satisfied if and only if the skew between any nonfaulty logical clocks is bounded.
- The *accuracy* condition is satisfied if and only if all nonfaulty logical clocks have a bounded drift with respect to real time.

Since COTS-based solutions are preferred within GUARDS, we focused on software-implemented algorithms. In particular, we considered both convergence averaging and convergence nonaveraging algorithms [57].

In a convergence averaging algorithm, each node resynchronizes according to clock values obtained through periodic one-round clock exchanges. On each node, the other clocks can be taken into account through a mean-like function [40] or a median-like function [46]. The worst-case skew of these algorithms is dominated by the uncertainty on transmission delay. They can tolerate f arbitrarily faulty nodes in a (fully connected) network of n nodes, under the sufficient condition that $n > 3f$.

In a convergence nonaveraging algorithm, each node periodically seeks to be the system synchronizer. To deal with possible Byzantine behavior, the exchanged messages can be authenticated [63]. The worst-case skew of these algorithms is dominated by the maximum message transit delay. When authentication is used for internode message exchanges, they can tolerate f arbitrarily faulty nodes with only $n > 2f$ nodes.

The GUARDS architecture uses a convergence-averaging solution based on [46] and applied to up to four nodes (i.e., ICN-managers in our architecture). This choice was motivated mainly by reasons of performance and design simplicity. It implies that the probability of occurrence of a Byzantine clock must be carefully evaluated in a three-channel configuration. This probability is expected to be very small, since the ICN serial links are broadcast media and the ICN-managers can check whether they receive a syntactically correct synchronization message in a well-defined local time window.

The global clock maintained by the set of ICN-managers is broadcast via the intrachannel back-plane busses, to the processors and I/O boards local to a channel.

[2] The exception is that of Byzantine clock behavior (see Section 4.1).

4.2 Interactive Consistency

The issue of exchanging private data between channels and agreeing on a common value in the presence of arbitrary faults is known as the interactive consistency problem (the symmetric form of the Byzantine agreement problem) [50]. The two fundamental properties that a communication algorithm must fulfill to ensure interactive consistency are:

- *Agreement*: If channels p and q are nonfaulty, then they agree on the value ascribed to any other channel.
- *Validity*: If channels p and q are nonfaulty, then the value ascribed to p by q is indeed p's private value.

In the general case, the necessary conditions to achieve interactive consistency in spite of up to f arbitrarily faulty channels are [39]:

- At least $3f+1$ channels.
- At least $2f+1$ disjoint interchannel communication links.
- At least $f+1$ rounds of message exchange.
- Bounded skew between nonfaulty channels.

Under the assumption of authenticated messages, which can be copied and forwarded but not undetectably altered by a relayer, the condition on the minimal number of channels can be relaxed to $f+2$. Nevertheless, at least $2f+1$ channels are still necessary if majority voting must be carried out between replicated application tasks.

The interactive consistency protocol used in GUARDS is based on the ZA algorithm [28], which was derived from the Z algorithm [64] by adding the assumption of authentication. In particular, authentication precludes the design fault in the Z algorithm identified in [45]. Following the hybrid fault model described in [45], the protocol allows for both arbitrarily faulty channels and channels affected by less severe kinds of faults (e.g., omission faults).

For performance reasons, and since by assumption the architecture only needs to tolerate accidental faults and not malicious attacks, we preferred to use a keyed checksum scheme for message authentication rather than resorting to true cryptographic signatures. Under this scheme, multiple checksums are appended to each (broadcast) message. Each checksum is computed over the concatenation of the data part of the message and a private key that is known only to the sender and to one of the broadcast destinations.

4.3 Scheduling

The ICN is scheduled according to a table-driven protocol. The schedule consists of a *frame* (corresponding to a given application mode) that is subdivided into *cycles* and *slots*. The last slot of a cycle is used for clock synchronization so the length of a cycle is fixed either by the required channel synchronization accuracy or by the maximum I/O frequency in a given mode. The other slots of a cycle are of fixed duration and can support one fixed-sized message transmission (and up to three message receptions). In the current implementation, each message may contain 1,000 bytes.

5 Interchannel Error Processing and Fault Treatment

From a conceptual viewpoint, it is common to consider fault tolerance as being achieved by error processing and fault treatment [2], [41]:

- *Error processing* is aimed at removing errors from the computation state, if possible, before failure occurrence. In general, error processing involves three primitives: error detection, error diagnosis, and error recovery.
- *Fault treatment* is aimed at preventing faults from being activated again and also involves three primitives: fault diagnosis, fault passivation, and reconfiguration.

In GUARDS, error recovery is achieved primarily by error compensation, whereby the erroneous state contains enough redundancy to enable its transformation into an error-free state. This redundancy is provided by active replication of critical applications (although diversification is not precluded) over the C channels; it is application-transparent and managed by software, including comparison or voting of computed results. Error processing thus relies primarily on N-modular redundancy to detect disagreeing channels and (when $C=3$) to mask errors occurring in the voted results at run-time. When $C=2$, two possibilities are offered, as already mentioned in Section 3.3:

- Error detection (locally by a channel) and compensation (by switching to a single channel configuration).
- Error detection (by channel comparison) and switching to a safe state (a degenerate form of forward recovery).

Fig. 2 illustrates the replicated execution of an iterative task in the case of a three-channel configuration. After reading the replicated sensors, the input values are consolidated across all channels after a two-round interactive consistency exchange over the ICN. The application tasks are then executed asynchronously, with preemptive priority scheduling allowing different interleavings of their executions on each channel. This diversifies the activities of the different channels, thereby allowing many residual design faults to be tolerated as if they were intermittents (cf. Section 2.2).

Application *state variables* (which contain values that are carried over between iterations) are used together with consolidated inputs to compute the output values which are exchanged in a single round over the ICN and voted. The voted results are then written to the actuators, possibly via output consolidation hardware, which allows the physical values to be voted.

Since neither the internal state variables of the underlying COTS operating systems nor the totality of the application state variables are voted, further error recovery is necessary to correct any such state that becomes erroneous (note that this may be case even in the event of a transient fault). However, this is a secondary, nonurgent error recovery activity since, until another channel is affected by a fault, the error compensation provided by output voting or switching can be relied upon to ensure that correct outputs are delivered to the controlled process.

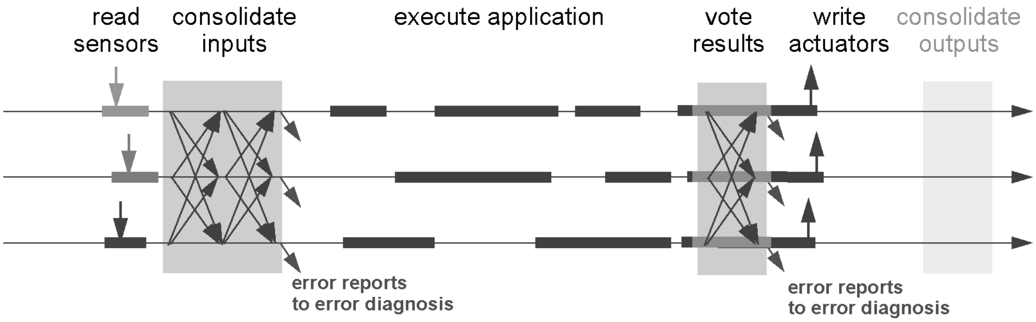


Fig. 2. TMR execution of an application function split in sequential threads.

Consequently, this secondary error recovery can be viewed as part of fault treatment.

In the next section, we describe the GUARDS diagnosis mechanisms, which include both *error* diagnosis, to decide whether the damage to a channel's state warrants further action, and *fault* diagnosis, to decide the location and type of the fault and, thus, the necessary corrective action.

Then, in Section 5.2, we describe the state recovery procedure that allows reintegration of a channel after a transient fault or repair of a permanent fault. Finally, Section 5.3 discusses mechanisms for output consolidation.

5.1 Diagnosis

The first step in diagnosis is to collect error reports generated during the interactive consistency and consolidation exchanges (majority voting discrepancies, timing errors, ICN bus transmission errors, protocol violations, etc.) and then to filter them to assess whether the extent of damage warrants further action. Indeed, some reported errors may not have resulted in any change to the state of a channel. Alternatively, if only a small part of the state has become erroneous, then an erroneous channel might correct itself autonomously by overwriting the erroneous variables during continued execution. If such fortuitous recovery does not occur, an explicit forward recovery action is necessary to reconstruct a correct state.

The filtering of errors is done using a software-implemented mechanism known as an α-count, which was originally proposed for the discrimination of transient versus intermittent-permanent faults [17]. Error reports are processed on a periodic basis, giving lower weights to error reports as they get older. A score variable α_x (initially set to 0) is associated to each component x to record information about the errors attributed to that component. The Lth judgment is accounted for as follows:

$$\alpha_x(L) = \alpha_x(L-1) + 1$$
$$\text{if component } x \text{ is perceived as faulty}$$
$$\alpha_x(L) = k \bullet \alpha_x(L-1)$$
$$\text{if component } x \text{ is perceived as correct (with } 0 < k < 1).$$

When $\alpha_x(L)$ becomes greater than a given threshold α_T, the damage to the state of component x is judged to be such that further diagnosis is necessary.

The appropriate filtering action can be provided by several different heuristics for the accumulation and decay processes (where $\alpha_x(L)$ takes slightly different expressions) [17], [55], [56]. For a given error distribution, the parameters of the heuristics can be determined through a dependability evaluation (for example, see [17]).

A distributed version of α-count is used in GUARDS to provide the error syndrome that is input to interchannel fault diagnosis. Each channel i maintains C α-count variables, one, α_{ii}, representing its opinion of its own health and C-1 variables, α_{ij}, $j \neq i$, representing its opinions of the health of the other channels. The α-counts are updated and processed cyclically. Each cycle N_α, called an α-cycle, has a duration chosen such that $N_\alpha \cdot n_1 = N_{frame}$, where n_1 is an integer and N_{frame} the duration of the ICN frame (see Section 4.3).

Since each channel may have a different perception of the errors created by other channels, the α-counts maintained by each channel must be viewed as single-source (private) values. They are consolidated at the end of each α-cycle through an interactive consistency protocol so that fault-free channels have a consistent view of the status of the instance (a consistent matrix A of α-count values). During the next α-cycle, fault diagnosis can thus be performed using A. The resulting diagnosis consists of a vector D whose elements D_i represent the diagnosed state of each channel (correct or requiring passivation and isolation).

The fault diagnosis problem has been extensively studied in the literature. An ideal diagnosis should be both *correct* and *complete*:

- A diagnosis is *correct* if any channel that is diagnosed as faulty is indeed faulty.
- A diagnosis is *complete* if all faulty channels are diagnosed as faulty.

In the current case, the interchannel tests have imperfect coverage so a channel requiring passivation is not necessarily accused by all correct channels [16], [43]. The algorithm in the current implementation diagnoses a channel as faulty if it is accused of being faulty by a majority of channels or, of course, if it accuses itself. This algorithm is correct and complete under the assumption that not more than one channel at a time is accused by a fault-free channel. However, due to the memory effect of the α-count mechanism, this assumption can be violated if near-coincident faults occur on different channels. In this situation, there is thus a trade-off between the probability of incorrect diagnosis caused by a long memory effect (high value of k) and the probability of having an incorrect majority vote due to slow elimination of a faulty channel (low value of *k*). This trade-off is the subject of ongoing research.

Once a channel has been diagnosed as requiring passivation, it is isolated (i.e., disconnected from the outside world) and reset (with the reinitialization of operating system structures). A thorough self-test is then carried out. If the test reveals a permanent fault, the channel is switched off and (possibly) undergoes repair. Whenever a channel passes the test (i.e., the fault was transient), or after having repaired a channel having suffered a permanent fault, it must be reintegrated to avoid unnecessary redundancy attrition.

It should be noted that the error filtering action of the α-count can effectively be turned off by setting its threshold $\alpha_T = 1$. In this case, any transient fault leading to a self-detected error (or to errors perceived by a majority of channels) will cause that channel to go through the possibly lengthy self-test and reintegration procedure, irrespective of the extent of the actual damage to the channel's state. A fault affecting another channel before reintegration of the former will induce a further decrease in the number of active channels. When transients are common, this policy can thus cause rapid switching to a safe state if the number of active channels becomes insufficient for error compensation to remain effective. The choice of whether filtering is used or, more generally, the value of the α-count threshold, thus leads to a classic trade-off between safety and reliability.

5.2 State Restoration

For a channel to be reintegrated, it must first resynchronize its clock, then its state, with the pool of active channels. Since not all state variables are necessarily consolidated through ICN exchanges, the state (or *channel context*) cannot be retrieved by simply observing the traffic on the ICN, but must be explicitly copied from the active channels. This is achieved by a system state restoration (SR) procedure, called *Running SR*, applied to the channel context, i.e., the set of application state variables whose values are carried over successive iterations without consolidation.

A minimum level of service must be ensured, even during the SR procedure, so a limited number of vital application tasks must be allowed to continue execution on the active channels. Running SR is therefore a multistep algorithm where, at each step, only a fraction of the state is exchanged. Furthermore, vital application tasks may update state variables while SR progresses.

The basic behavior of Running SR is the following (more details with variations and optimizations are given in [18], [19]). The channel context is arranged in a single (logical) memory block managed by a "context object." When the state of channel needs to be restored, the system enters an "*SR mode*." The *C*-1 active channels enter a "*put state*" submode while the joining channel enters a "*get state*" submode.

To take advantage of the parallel links of the ICN, the whole block of memory storing the channel context is split into *C*-1 subblocks of similar size, each managed by one of the active channels. Each active channel *i* propagates to the joining channel any updates to state variables belonging to block *i*. A *Sweeper* task is executed to transfer the *i*th block of the context. In the joining channel, transferred data are received and processed by a *Catcher* task. This task has most of the CPU time available since no application tasks are executed on that channel.

Switching from normal computation to the SR mode occurs at the beginning of an ICN frame, with a corresponding change in task scheduling, and SR completion always occurs at the end of a frame. After completion, signatures of the entire channel state are taken in each channel and exchanged through the interactive consistency protocol. State restoration is considered successful if all signatures match. Normal application scheduling is then reactivated on the next frame.

Since a deterministic, finite time is required to copy the memory block and any updates to already copied state variables are immediately propagated, the whole (parallel) state restoration is performed in a deterministic, finite time. The state restoration tasks are assigned a priority and a deadline and, for schedulability analysis, are treated the same as vital application tasks. Note that, during SR, the ICN has to support: 1) the normal traffic generated by the vital (i.e., nonstoppable) applications, 2) the extra traffic due to state variable updates, and 3) the traffic generated by the Sweeper task. SR will therefore normally require a mode change to suspend nonvital application tasks so as to release processor time and ICN slots for SR execution and communication.

5.3 Output Data Consolidation

The purpose of the output data consolidation system (cf. Fig. 1) is to map the replicated *logical outputs* of each channel onto the actual *physical outputs* to the controlled process, in such a way that the latter are either error-free or in a safe position. Such consolidation, placed at the physical interface with the controlled process, is the ultimate error confinement barrier and is a complement to any software-implemented voting of the logical outputs.

A given instance of the architecture could have several different output consolidation mechanisms according to its various interfaces with the controlled process. Ideally, an output data consolidation mechanism should extend into the controlled process itself to prevent the physical interface to the process from becoming a single point of failure. A typical example would be a control surface (e.g., in a fly-by-wire application) that can act as a physical voter by summing the forces produced by redundant actuators. Alternatively, a single channel can be designated to control a given actuator. Failures of that actuator can be detected at the application level by means of additional sensors allowing each channel to read back the controlled process variable and check it against the requested output. Recovery can then be achieved by switching to an alternative actuator. Other process-specific output data consolidation mechanisms used in the GUARDS end-user application domains include combinatorial logic implemented by relay or fluid valve networks, and the "arm-and-fire" technique commonly used to trigger space vehicle pyrotechnics (one channel sends an "arm" command, which is checked by the other channels, then all channels send matching "fire" commands; the pyrotechnics are triggered only if a majority of the latter concur with the former).

Output consolidation mechanisms such as these may be used for various end-user instances of the architecture. By definition, such process-specific techniques cannot be generic so no specific research has been carried out in this direction. However, the project has considered generic output consolidation mechanisms for networked and discrete outputs. A prototype consolidation mechanism is being implemented for discrete digital or analog outputs that can be electrically isolated from each other and then connected through a wired-OR to the output devices. Consolidation is achieved by having each channel read back its own output and those of the other channel(s) so that a vote can be carried out in software. Each channel then sends selection signals to a hard-wired voter (one per channel) that allows or disconnects that channel's outputs. This approach relies on the assumption that the output devices can tolerate the short but inevitable output glitch caused by the read-back, vote, and disconnection delay.

6 Real-Time Scheduling

The architecture is capable of supporting a range of scheduling models (cf. Section 2.3). In this section, we focus on the standard preemptive priority-based scheme. We also discuss the consequences on scheduling of the ICN network.

Our timing analysis is based upon the Response-time Analysis [7], [44]. We assume that any communication between applications is asynchronous through the shared memory. The use of round-robin scheduling on the intrachannel VME bus allows all shared-memory accesses to be bounded. This is adequate because it is assumed that the number of hosts within a channel is small. Furthermore, we assume the use of a nonblocking algorithm such as that proposed in [62] to avoid the problems associated with remote blocking.

6.1 Interchannel Replication of Applications

For an application task to be replicated, it must behave deterministically and each replica task must process the same inputs in the same order. At any point where there is potential for replica divergence, the channels must perform an interactive consistency agreement. Unfortunately, the cost of executing interactive consistency agreement protocols can be significant. There is, therefore, a need to keep their use to a minimum.

In our approach, we force all replicated tasks to read the same internal data. We can thus trade off fewer agreement communications (and therefore greater efficiency) against early detection of errors. If we assume that each replica does not contain any inherently nondeterministic code, replica determinism and error masking (or detection) can be ensured by:

- Performing interactive consistency agreement or Byzantine agreement on single-sourced data.
- Ensuring that all replicas receive the same inputs when those inputs are obtained from other replica tasks (replicated inputs).
- Voting on any vital output.

6.1.1 Agreement on Sensor Inputs

To reduce the complexity of the input selection algorithm, which processes the vector of redundant values consolidated through the interactive consistency exchange, it is important to minimize the error between the redundant input values. However, since the tasks are independently scheduled on each channel, they could read their corresponding sensors at significantly different times. This is similar to the input jitter problem where a task (τ) implementing a control law has to read its input on a regular basis. If jitter is a problem, the solution is to split the task into two tasks (τ^{ip},τ'). τ^{ip} has a release time[4] and a deadline appropriate for the dynamics (and the allowable jitter) of the physical quantity being measured by the sensor. Task τ' has the original τ's deadline and is executed at an offset from the release time of τ^{ip}. We will discuss what value this offset should have in Section 6.2.

6.1.2 Identical Internal Replicated Input

Two cases need to be considered when reader and writer tasks share the same data, according to whether or not there is an explicit precedence constraint between the writer and the reader. When there is such a constraint, then it can be captured by the scheduling. When tasks share data asynchronously (and, therefore, there is no explicit precedence constraint between the writer and the reader), there are four types of interaction:

- Periodic writer—Periodic reader: the periods of the two tasks do not have a simple relationship.
- Periodic writer—Sporadic reader: There is no relationship between the period of the writer and the release of the reader.
- Sporadic writer—Sporadic reader: There is no relationship between the release of the writer and the release of the reader.
- Sporadic writer—Periodic reader: There is no relationship between the release of the writer and the period of the reader.

In all of these cases, to ensure each replica reads the same value, we keep more than one copy of the data (usually two is enough) and use timestamps [9], [51]. The essence of this approach is to use off-line schedulability analysis [7] to calculate the worst-case response times of each replicated writer. The maximum of these values is added to the release time of the replicas (taking into account any release jitter) to give a time by which all replicas must have written the data (in the worst case). To allow for clock drift between replicas, the maximum skew, ε, is also added. This value is used as a timestamp when the data is written.

A reader replica simply compares its release time with the data timestamp. If the timestamp is earlier, then the reader can take the data. If the timestamp is later than its release time, then the reader knows that its replicated writer has potentially executed before the other replicated writers. It must therefore take a previous value of the data (the most recent) whose timestamp is earlier than its release time. All

[4] We assume that all I/O is periodic in nature.

reader replicas undertake the same algorithm and consequently get the same value.

6.1.3 Output Voting

Where output voting is required, it is again necessary to transform the replicated task writing to the actuators into two tasks (τ' and τ^{op}): τ' sends the output value across the ICN for voting, and τ^{op} reads the majority vote and sends this to the actuator. The deadline of τ' will determine the earliest point when the ICN manager can perform the voting. The offset and deadline of τ^{op} will determine when the voted result must be available and the amount of potential output jitter. Hence, the two tasks have similar timing characteristics to the tasks used for input agreement (cf. Section 6.1.1). The main difference is that there is a simple majority vote rather than an agreement protocol involving three separate values.

6.2 Handling Offsets

A real-time periodic transaction model has been developed in which periodic transaction *i* consists of three tasks τ_i^1, τ_i^2, and τ_i^3. Task τ_i^1 reads a sensor and sends the value to the ICN manager. Task τ_i^2 reads back from the ICN manager the set of values received from all the replicas; it consolidates the values and processes the consolidated reading and eventually produces some data. It sends this data for output result consolidation to the ICN manager. Task τ_i^3 reads the consolidated result from the ICN manager and sends it to the actuator.

This form of real-time transaction is implemented by timing offsets. Analysis of task sets with offsets is NP complete [44] and even suboptimal solutions are complex [6], [8], [65]. The approach we take is based on [10], modified to take into account the fact that the computational times of τ_i^1 and τ_i^3 (respectively C_i^1 and C_i^3) are much smaller than C_i^2, the computational time of τ_i^2, i.e., $C_i^2 >> \max(C_i^1, C_i^3)$.

Once offsets has been assigned, a check must be made to ensure that: 1) the response times of the individual tasks are less than the offsets of the next task in the transaction, 2) there is enough time before the offset and after the response to transmit data on the ICN network, and 3) that the deadline of the transaction has been met. If any of these conditions is violated, then it may be possible to modify the offsets of the transaction violating the condition in an attempt to satisfy all the requirements [10].

6.3 Scheduling the ICN Network

Following the individual schedulability analysis of each channel, the following characteristics are known for each task participating in replicated transactions:

- Period
- Response-time
- Offset
- Deadline

The ICN tables can be built from this information—in the same way as cyclic executive schedules can be constructed [22]. Since all communication through the channels' shared memory is asynchronous, the ICN manager can take the data any time after the producing task's deadline has expired.

Of course, there is a close relationship between the scheduling of the channels and the scheduling of the ICN network. If the off-line tool fails to find an ICN schedule, it is necessary to revisit the design of the application.

7 Architecture Development Environment

The generic architecture is supported by an Architecture Development Environment [49] consisting of a set of tools for designing instances of the architecture according to a coherent and rigorous design method. The toolset allows collection of the performance attributes of the underlying execution environment and the analysis of the schedulability of hard real-time threads, not only within each processing element of the system, but also among them. This allows in particular a rigorous definition of critical communication and synchronization among the redundant computers.

7.1 Design Method

The design and development of a GUARDS software application are centered on a hard real-time (HRT) design method, which allows real-time requirements to be taken into account and verified during the design. The method also addresses the problem of designing replicated, fault-tolerant architectures, where a number of computing and communication boards interact for the consolidation of input values and output results.

The design of a GUARDS application is defined as a sequence of correlated activities that may be reiterated to produce a software design that complies with both the functional and nonfunctional requirements of the application. Three design activities are identified:

- *Functional* architecture design, where the software application is defined through an appropriate design method and according to its functional requirements and its performance requirements (task periods, deadlines, etc.).
- *Infrastructure* architecture design, where the required hardware boards and generic GUARDS software components are identified. They constitute the underlying computing environment of the application software.
- *Physical* architecture design, where the functional architecture is mapped onto the infrastructure and analyzed according to the performance requirements. This is done not only for the processors within each replicated channel, but also at the interchannel level, to determine the ICN exchanges needed to consolidate input values and output results.

7.2 Interchannel Schedulability

According to the dependability requirements, each critical application task replica needs to consolidate its inputs and its output results with those of the corresponding replicas on the other channels (Fig. 3). Each application task τ_i is structured as a real-time transaction consisting of three subtasks, or threads, τ_i^1, τ_i^2, and τ_i^3 responsible for input

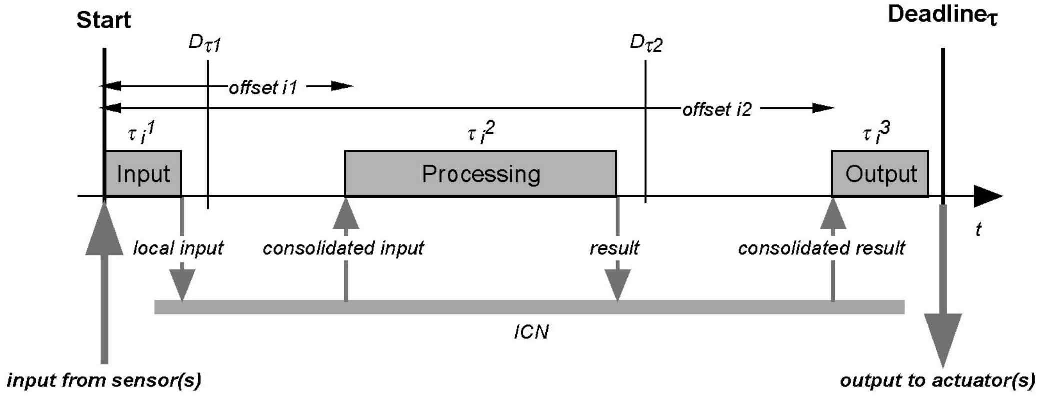


Fig. 3. Decomposition of critical tasks.

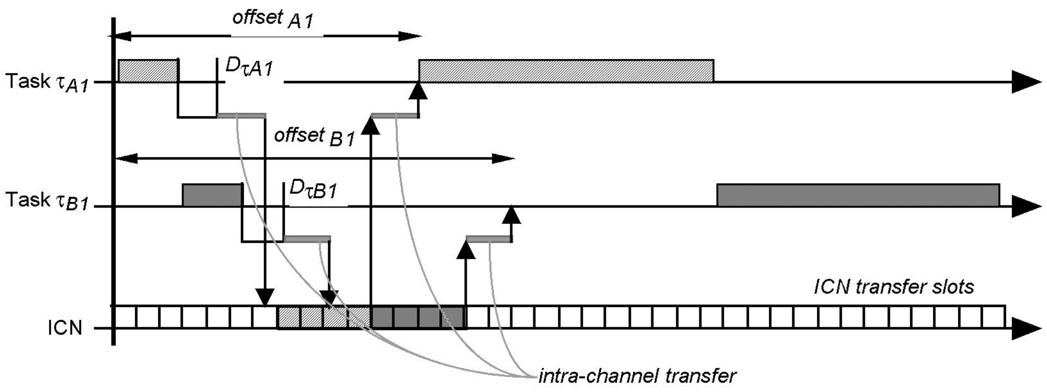


Fig. 4. Interchannel schedulability.

acquisition, result calculation and output actuation (cf. Section 6.2).

For each application task τ_i, a deadline is set that defines the time by which the final value must be sent to the actuator(s) (corresponding to the third thread of the transaction). Intermediate deadlines $D_{\tau 1}$ and $D_{\tau 2}$ are also introduced for the first and second threads. They define the time by which the input or output results are (or must be) ready for transfer through the ICN (after a fixed intrachannel transfer time) and consolidated. The transfer and consolidation of each value over the ICN must take place at predefined transfer slots (to synchronize such activities on each channel) and the needed duration determines an offset for the activation of the following thread (Fig. 4).

Although the final deadline is set by the requirements, intermediate deadlines can be set arbitrarily during the design, according to the intrachannel schedulability analysis and the allocation of ICN transfer slots. Different intermediate deadlines may imply different ICN transfers (i.e., slot allocations) and consequently different offsets. Consequently, the HRT schedulability analysis (at the intrachannel level) must take into account the possible tuning of HRT design attributes (i.e., intermediate deadlines and offsets), as well as the slot allocation (i.e., the interchannel schedulability).

7.3 Supporting Tools

The *functional* architecture design is supported by an appropriate method and tool. To provide for genericity, GUARDS does not force the selection of a specific method, but it is assumed that the method selected by the user is indeed suitable for the design of real-time software systems. Nevertheless, a survey and an analysis of design methods have shown that only HRT-HOOD [23] addresses explicitly the design of *hard* real-time systems, providing means for the verification of their performance. Therefore, HRT-HOOD was selected as the baseline design method and HRT-HoodNICE adopted as supporting tool [34].

However, the analysis also revealed several weaknesses of the method, in particular related to the design of *distributed* systems. The method was thus extended to include the concept of *Virtual Nodes*, similar to that in the HOOD 3.1 method [31]. The extended method can take into account the lane dimension of GUARDS (by allocating objects to different processors within a channel) and the integrity dimension (by defining spatial firewalls around objects of a given criticality). The HRT-HoodNICE toolset has been accordingly enhanced.

The *infrastructure* architecture design is supported by a specific toolset that manages an archive of hardware and software components. Such components are described by their relations, compatibilities, and performance attributes.

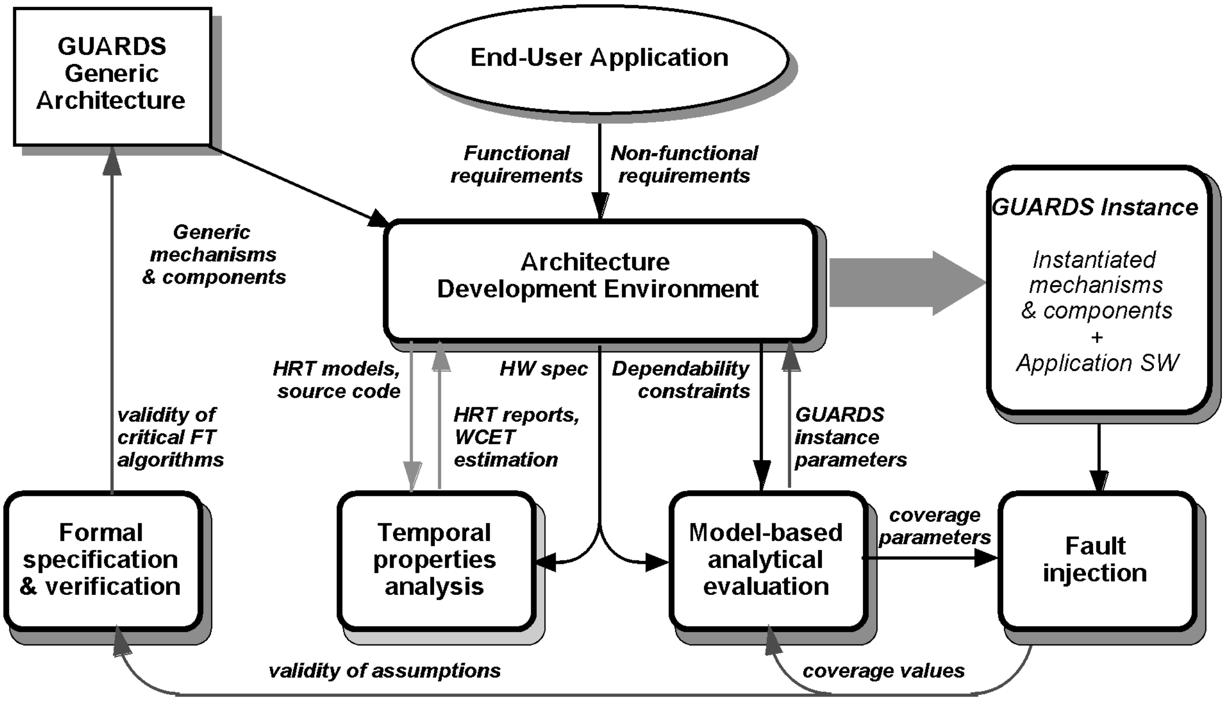

GUARDS Generic Architecture
End-User Application
Functional requirements
Non-functional requirements
Generic mechanisms & components
Architecture Development Environment
GUARDS Instance
Instantiated mechanisms & components + Application SW
HRT models, source code
HW spec
Dependability constraints
validity of critical FT algorithms
HRT reports, WCET estimation
GUARDS instance parameters
Formal specification & verification
Temporal properties analysis
Model-based analytical evaluation
coverage parameters
Fault injection
validity of assumptions
coverage values

Fig. 5. Main interactions between architecture development and validation.

The tool selects the needed components according to the characteristics of the required GUARDS instance.

As part of the *physical* architecture design, the application tasks (i.e., HRT objects) identified in the functional architecture are mapped onto the infrastructure architecture. They are coupled with the real-time models of the selected components in order to analyze and verify their schedulability properties. This is done by the Temporal Properties Analysis toolset, which analyzes the performance of the resulting distributed software system.

The Temporal Properties Analysis toolset includes a Schedulability Analyzer and a Scheduler Simulator, based on those available in HRT-HoodNICE. They have been enhanced to provide a more precise and realistic analysis (by taking into account the concept of thread offsets) and to cope with the specific needs of a redundant fault-tolerant architecture (by allowing the analysis of the interactions over the ICN).

A further result of the physical architecture design is that, on the basis of the real-time models produced by the verification tools, the critical interactions among software functions on different channels are scheduled in a deterministic way. The ICN transfer slots allocated to them and a set of predefined exchange tables are produced automatically.

As a final step of the design phase, the overall structure of the software application is extracted from the HRT-HOOD design and the related code is automatically generated. To this end, a set of mapping rules has been defined to translate the HRT-HOOD design in terms of threads implemented in a sequential programming language (which could be C or the sequential subset of Ada) and executed by a POSIX compliant microkernel [68].

8 Validation

The validation strategy implemented within GUARDS has two main objectives [3]:

- A short-term objective: the validation of the design principles of the generic architecture, including both real-time and dependability mechanisms.
- A long-term objective: the validation of the development of instances of the architecture implementing specific end-user requirements.

A large spectrum of methods, techniques, and tools has been considered to address these validation objectives and to account for the validation requirements expressed by the emerging trans-application domain standard IEC 1508 [33].

Following the comprehensive development model described in [42], the validation strategy is closely linked to the design solutions and the proposed generic architecture. The validation environment that supports the strategy includes components for verification and evaluation, using both analytical and experimental techniques. Fig. 5 illustrates the relationship between the components of the validation environment and their interactions with the architecture development environment.

Besides the three main validation components (namely, formal verification, model-based evaluation, and fault injection), the figure explicitly identifies the role played by the methodology and the supporting toolset being developed for schedulability analysis (cf. Section 7.3). The figure also depicts the complementarity and relationships among the three validation components. In particular, fault injection (carried out on prototypes) complements the other validation components by providing means for: 1) assessing the validity of the necessary assumptions made by the formal verification task, and 2) estimating the coverage

TABLE 1
Formal Verification Approaches

Target Mechanism	**Clock Synchronization**	**Interactive Consistency**	**Fault Diagnosis**	**Multi-level Integrity**
Properties Verified	Agreement Accuracy	Agreement Validity	Correctness Completeness	Segregation Policy (Multi-level Objects)
Approach	Theorem Proving	Model Checking		
Description and Specification	Higher Order Logic	Process Algebra (CCS) and Temporal Logic (ACTL)		
Supporting Tool	PVS	JACK		

CCS: Calculus for Communicating Systems ACTL: Action based version of Computation Tree Logic

parameters included in the analytical models for dependability evaluation. The following three subsections briefly describe the related validation activities.

8.1 Formal Verification

Formal approaches were used both for specification and as a design-aid. We concentrated our effort on four dependability mechanisms, which constitute the basic building blocks of the architecture: 1) clock synchronization, 2) interactive consistency, 3) fault diagnosis, and 4) multilevel integrity.

The formal approaches that have been applied included both theorem proving and model checking. Table 1 summarizes the main features of the verifications carried out for each of the target mechanisms.

The work carried out on the verification of clock synchronization relied heavily on PVS (*Prototype Verification System*) [48]. It led to the development of a general theory for averaging and nonaveraging synchronization algorithms [60]. The verification of the synchronization solution used in GUARDS (cf. Section 4.1) was derived as an instantiation of this general theory.

The verifications concerning interactive consistency [12], [14], fault diagnosis [13], and multilevel integrity [27], [61] were all based on model checking using the JACK (*Just Another Concurrency Kit*) toolset [21]. This integrated environment provides a set of verification tools that can be used separately or in combination. Due to the complexity of the required models, the toolset was extended to include a symbolic model checker for ACTL [26].

These studies demonstrated the feasibility and the benefits of formal methods on realistic industrial problems using state-of-the-art tools. We believe this is an important outcome that can significantly facilitate the acceptance of the GUARDS generic architecture for critical applications. It is also expected that further exploitation of the complementarity between theorem proving and model checking could facilitate a wider industrial acceptance of formal methods.

8.2 Dependability Evaluation

Model-based dependability evaluation is widely recognized as a powerful means to make early and objective design decisions by assessing alternative architectural solutions. Nevertheless, fault-tolerant distributed systems (such as GUARDS instances) pose several practical modeling problems ("stiffness," combinatorial explosion, etc.). Moreover, due to the variety of the application domains being considered, the dependability measures of interest encompass reliability, availability, and safety.

To cope with these difficulties, we adopted a divide-and-conquer approach, where the modeling details and levels are tailored to fit the needs of the specific evaluation objectives. This was achieved by first focusing the modeling effort either on generic or specific architectural features, or on selected dependability mechanisms. Then, an abstract modeling viewpoint that aims to provide a global framework for configuring instances to meet specific application dependability requirements was devised. Finally, elaborating on previous related work (e.g., [36]), a detailed modeling viewpoint that supports incremental and hierarchical evaluation has been considered.

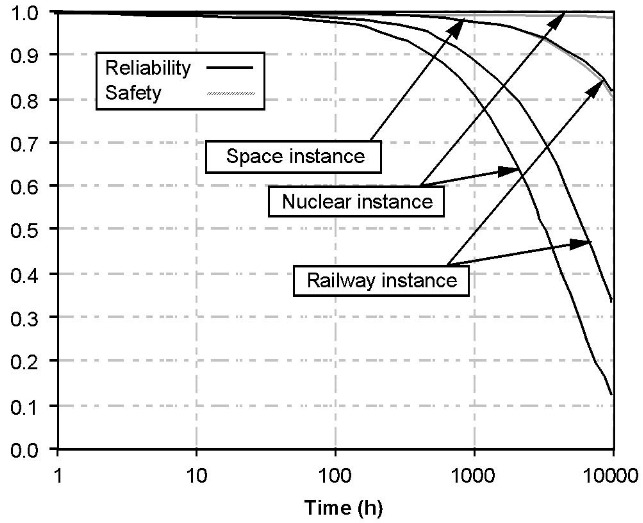
Table 2 identifies the various dependability evaluation activities carried out according to these three modeling viewpoints.

The focused models addressed several issues concerning the analysis of generic mechanisms (e.g., α-count [17]) and of specific features for selected instances (phased missions, for the space prototype instance [20], intrachannel error detection for the railway prototype instance).

The second viewpoint aims to establish a baseline set of models for the three instances of the architecture described in Section 9 (see also [54]). A general notation is introduced that allows for a consistent interpretation of the model parameters (layers, correlated faults, etc.) for each prototype instance. This work provides the foundation of a generic modeling approach to guide the choice of a particular instantiation of the architecture, according to the dependability requirements of the end-user application. A large number of parameters (proportion of transient vs. permanent faults, correlated faults in the hardware and software layers, coverage factors, error processing rates, etc.) have

TABLE 2
Dependability Evaluation Viewpoints and Studies

Modeling Level	Focused	Abstract	Detailed
Targeted Mechanisms, Strategies, Instances	- α-count mechanism - Phased missions - Intra-channel error detection mechanism	- Railway prototype Instance - Nuclear submarine prototype Instance - Space prototype instance	Overall design and interactions (Nuclear submarine prototype instance)
Formalism	Stochastic activity networks and generalized stochastic Petri nets	Generalized stochastic Petri nets	Stochastic Petri nets
Supporting Tools	UltraSAN, SURF-2	SURF-2	MOCA-RP
Resolution Methods	Analytical and Monte-Carlo simulation	Analytical and method of stages	Monte-Carlo simulation



Parameters	Railway	Nuclear	Space
Failure rates (h^{-1})			
Hardware components			
ICN manager (10^{-5})			
- *independent*	.999 10^{-5}	.999 10^{-5}	.999 10^{-5}
- *correlated (interchannel)*	10^{-8}	10^{-8}	10^{-8}
Processor (10^{-5})			
- *independent*	.99 10^{-5}	.989 10^{-5}	.989 10^{-5}
- *correlated (intrachannel)*	0	0	0
- *correlated (interchannel)*	10^{-7}	10^{-7}	10^{-7}
- *correlated (global)*	0	10^{-8}	10^{-8}
Software layers:			
correlated (interchannel) only			
Executive	10^{-4}		
System	10^{-6}		
Application (integrity lev. 1)	10^{-5}		
Application (integrity lev. 2)	10^{-6}		
Coverage factors			
ICN manager	.80		
Processor & software layers	.80 (Except, Nuclear .99)		
Inter-channel voting	1		
Integrity policy	.80		
Self-diagnosis	.99		

Fig. 6. Comparison of the dependability of selected instances.

been included in the models, allowing intensive sensitivity analyses to be carried out. As an example of the results obtained, Fig. 6 compares reliability and safety for the three instances considered,[5] for one set of values of the parameters included in the models. The ranking of the reliability curves simply reflects the redundancy at the channel level (C = 4, 3, and 2 for the space, railway, and space instances, respectively). However, in the case of safety, the ranking of the nuclear and railway instances is reversed. This is mainly due to the fact that, in the nuclear instance, correlated design faults in either lane of the executive layer can be detected by the interlane comparison within each channel.

Detailed models are needed to allow for a more comprehensive analysis of the behavior of the instances (dependencies, error propagation, etc.). Specific work has addressed hierarchical modeling with the aim of mastering the complexity of such detailed models [35]. This work is directed mainly at: 1) enforcing the thoroughness of the analysis, 2) helping the analyst (i.e., a design engineer who is not necessarily a modeling expert). It is currently being applied and refined on the nuclear submarine prototype instance.

Although they were supported by different tools, namely UltraSAN [59], MOCA-RP [25] and SURF-2 [11], the modeling efforts all rely on the stochastic Petri net formalism. This should facilitate reuse of the results of the various studies (both models and modeling methodology).

[5] As safety is only a minor concern for the application targeted by the space instance, only the reliability curve is shown for that instance.

TABLE 3
Fault Injection-Based Testing of the GUARDS' Dependability Mechanisms

	Fault/Error Type	Trigger Event	Observation
ICN Mechanisms: - Clock Synchronization - Interactive Consistency - α-count and Diagnosis - ...	- Omit/Delay Synch. Mess. - Alter ICN Table - Repeat the Same Error	Specific Frame/Cycle/Slot	ICN Status Vectors
Host Mechanisms: - Multi-level Integrity - Control Flow Monitoring - HRT Scheduling - ...	- Issue a Forbidden Call - Provoke Illegal Branch - Modify Task Priorities	Condition (Flag/Counter) on the Host	- Integrity Kernel Activity - Control Flow Monitor - ICN Status Vectors

8.3 Fault Injection

The main objectives for the planned fault injection activities are twofold: 1) to complement the formal verification of GUARDS mechanisms (i.e., removal of residual deficiencies in the mechanisms), and 2) to support the development of GUARDS instances by assessing its overall behavior in the presence of faults, in particular by estimating coverage and latency figures for the built-in error detection mechanisms [4].

Indeed, as an experimental approach, fault injection provides a pragmatic means to complement formal verification by overcoming some of the behavioral and structural abstractions made, especially regarding the failure mode assumptions. Fault injection is to be carried out on complete prototypes so the mechanisms are tested globally when they have been integrated into an instance. In particular, the interactions between the hardware and software features are taken into account.

Although available tools could have been used—albeit with some extensions—a specific fault injection toolset (FITS) is being developed. Such a toolset is a major feature of the validation environment made available to support the end-users in the development of specific instances of the generic architecture.

Both for cost-effectiveness and flexibility, the fault injection environment is based on the software-implemented fault injection (SWIFI) technique [32]. This also allows tests to be conducted more efficiently, since: 1) a limited number of errors can simulate the consequences of a large number of faults, 2) it is less likely that the injected error fails to exercise the dependability mechanisms.

Two main levels of injection are being considered, whether the targeted mechanisms are implemented by the ICN-manager board or by the intrachannel processors. In practice, the implementations differ significantly: Whereas fault injection on the intrachannel processors can be assisted by the resident COTS operating systems and debug facilities [24], [38], the ICN-manager only has a very simple cyclic executive.

We concentrate here on the verification objective, which is the main focus of the current implementation of FITS; further features are needed to address the evaluation objective (e.g., see [5]). Some examples of the experiments aimed at testing various mechanisms are given in Table 3.

Besides injecting specific fault/error types, FITS allows injection to be synchronized with the target system by monitoring trigger events. Of course, the observations depend on the targeted mechanisms. While it is primarily intended to inject on a single channel, observations are carried out on all channels. Initial experiments will focus on the ICN mechanisms.

9 Prototypes

Several practical instances of the generic architecture have been studied and a prototype for each of the three end-user domains is under development. The basic building blocks are practically identical in each instance. However, the configurations of the instances are very different and offer quite different fault tolerance strategies. Moreover, although the operating systems chosen by each end-user are POSIX-compliant, they are not identical; neither are the end-users' preferred system development environments. Consequently, although there is a single specification of the generic software components of the fault-tolerant and integrity management layer, they have different practical instantiations in each instance.

9.1 Railway Instances

One instance studied for the railway domain is a fairly classic triple modular redundant (TMR) architecture with one processor per channel (Fig. 7). If a channel is diagnosed to be permanently faulty, the system degrades to a two-out-of-two mode. If a fault should occur while in this mode, the instance is switched to a safe state if the errors caused by the fault are detected (either locally within a channel or by two-out-of-two comparison).

This instance would employ Motorola 68040 or 68360 processors, each running a POSIX-compliant VxWorks operating system. Compared to currently deployed systems, the innovative aspect of this instance is the co-existence of two levels of application software integrity corresponding to very different degrees of criticality:

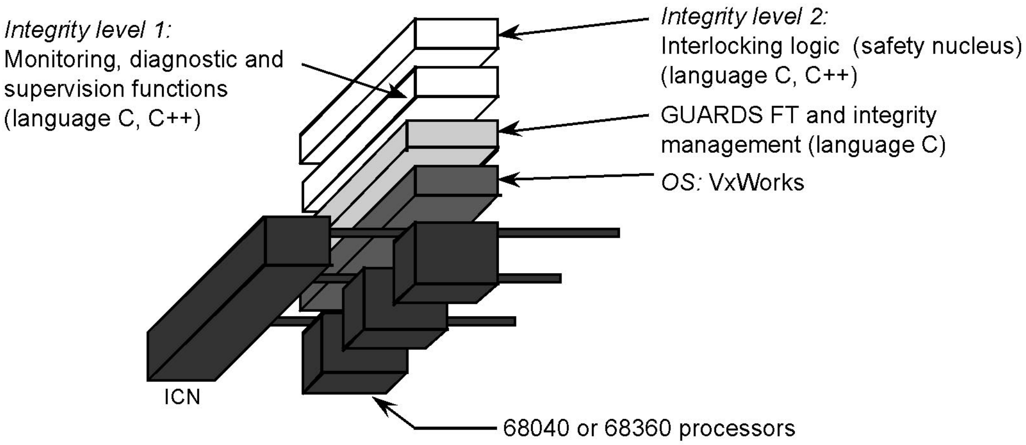

Integrity level 1:
Monitoring, diagnostic and supervision functions (language C, C++)
Integrity level 2:
Interlocking logic (safety nucleus) (language C, C++)
GUARDS FT and integrity management (language C)
OS: VxWorks
ICN
68040 or 68360 processors

Fig. 7. Railway triplex instance ($C = 3$, $M = 1$, $I = 2$).

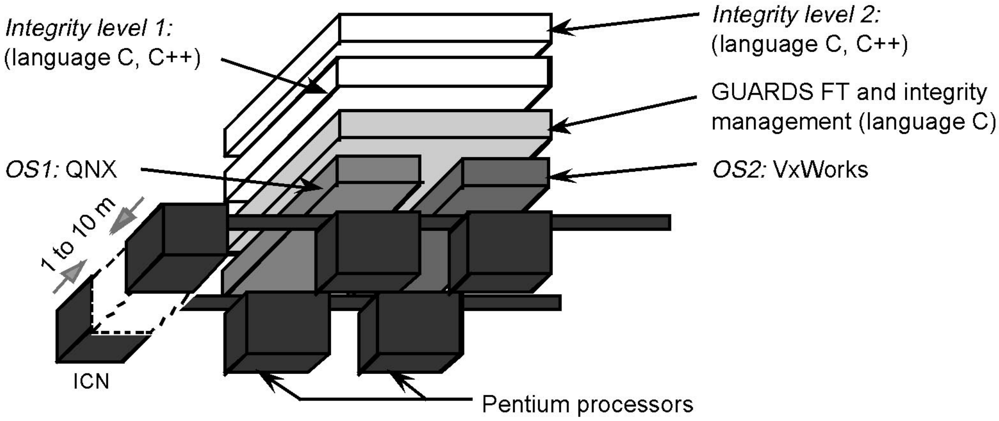


Fig. 8. Nuclear submarine duplex instance ($C = 2$, $M = 2$, $I = 2$).

- Highly critical interlocking logic or safety nucleus, which must be the highest integrity.
- Noncritical monitoring, diagnostic, and supervision functions.

This is a significant departure from current practice in railway applications, where these two levels of criticality would normally be implemented on separate instances. However, there is an appreciable economic advantage to be gained when it is possible to share the same hardware between both levels (e.g., for small railway stations).

A second railway instance has also been considered for an embedded train control system and is currently being prototyped. This is a straightforward duplex fail-safe configuration.

9.2 Nuclear Submarine Instance

The targeted nuclear submarine application is a secondary protection system. The instance considered for this application is a dual-channel architecture with two Pentium processors in each channel (Fig. 8). To prevent common-mode failures of both channels due to physical damage, the channels are geographically separated by a distance of several meters. Like the railway triplex system, this instance hosts two levels of integrity.

An innovative aspect of this instance is the use of two processors in each channel, with two different POSIX-compliant operating systems: QNX and VxWorks. Apart from the operating systems, both processors in each channel run identical application software. The copies of application components executing on each lane form self-checking pairs to provide detection of errors due to faults activated independently on each lane. In particular, this includes physical faults (of the processors) and design faults of the processors and their operating systems. It is assumed that design faults of the operating systems are activated independently, based on the fact that their designs are diversified. Although the processors are identical, we also assume that faults in their design will be independently activated, based on their diversification of utilization (due to loose coupling and diversification of operating systems).

As long as both channels are operational, they operate in a two-out-of-two mode. Results of computations that are declared as error-free by the intrachannel mechanisms are compared and, in case of disagreement, the instance is put into a safe state. However, if errors are detected locally, by intrachannel mechanisms, the channel declares itself to be faulty and the instance switches to single channel operation. Note that this strategy is different from that of the

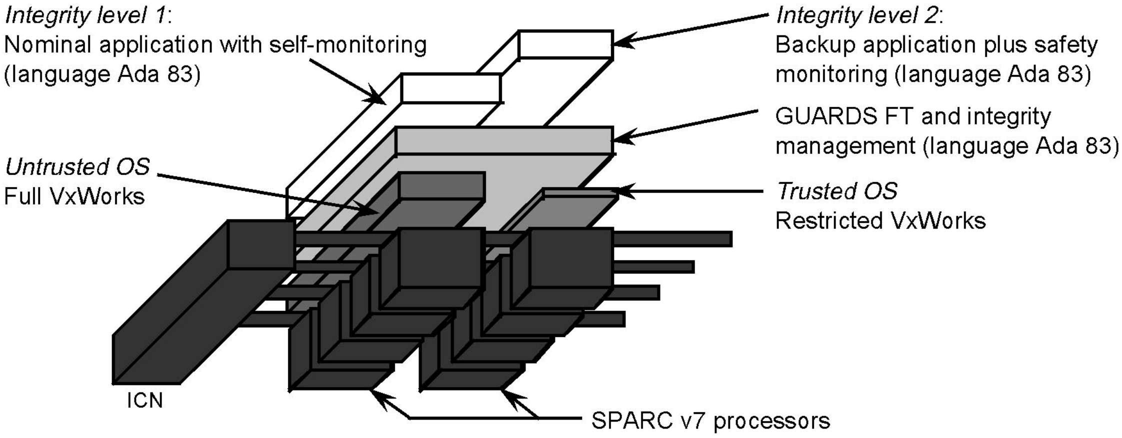


Fig. 9. Space quadruplex instance ($C = 4$, $M = 2$, $I = 2$).

two-channel configurations of the railway instances (duplex instance or triplex instance degraded to duplex), which switch to a safe state whether the error is detected locally or by comparison.

9.3 Space Instance

The instance of the architecture for space applications is the most complex of those considered. It is a full four-channel instance of the architecture capable of tolerating arbitrary faults at the interchannel level (Fig. 9). Degradation to three-, two-, and one-channel operation is possible. This instance also features two levels of integrity.

Like the instance intended for the nuclear submarine application, this instance also possesses two lanes, but for a different reason. For the nuclear application, the aim was to allow diversified but equivalent operating systems to be used so that errors due to design faults could be detected. Here, the objective is to have one of the lanes (the *secondary* lane) act as a back-up for the other lane (the *primary* lane). Each lane supports a different operating system and different application software:

- The primary lane runs a full-functionality version of VxWorks and a *nominal* application that provides full control of the spacecraft and its payload. The application includes built-in self-monitoring based on executable assertions and timing checks.
- The secondary lane runs a much simpler, restricted version of VxWorks and a safety-monitoring and simple back-up application. The purpose of the latter is to provide control of the spacecraft in a very limited "survival" mode (e.g., sun-pointing and telemetry/telecontrol functions).

The idea is that neither the full VxWorks nor the nominal application supported by the primary lane can be trusted to be free of design faults. However, the restricted version of VxWorks and the application software supported by the back-up lane are assumed to be free of design faults and thus trustable. The aim is to allow continued (but severely degraded) operation in the face of a correlated fault across all processors of the primary lane. Errors due to such a correlated fault can be detected in two ways:

- By the self-monitoring functions (essentially control-flow monitoring and executable assertions) included within the nominal application.
- By a safety-monitoring application executed by the secondary lane while the primary lane is operational.

In view of the differing levels of trust of the applications supported by the primary and secondary lanes, they are placed at different levels of integrity. The nominal application (on the primary lane) is not trusted, so it is assigned to the lower integrity level. The back-up application is assumed to be free of design faults and is placed at the higher integrity level. This separation of the integrity levels on different lanes provides improved segregation (firewalling) between the two levels of integrity.

10 Conclusions and Future Work

The GUARDS project is an ambitious one. We have defined a generic fault-tolerant architecture based on COTS components and a small set of purpose-designed hardware and software building blocks. This architecture can be configured along three different dimensions (channels, lanes, integrity levels) to meet the dependability requirements of a wide variety of end-user applications.

The design of the architecture has shamelessly borrowed ideas from previous work (in particular, SIFT [47], MAFT [37], FTPP [30], and Delta-4 [52]). Like SIFT and Delta-4, the focus has been on software-implemented fault tolerance, with a minimum of special-purpose hardware. Like SIFT and MAFT, the architecture uses a fully connected broadcast bus network for interchannel communication. Furthermore, the architecture uses the ZA algorithm for interactive consistency [28], which resulted from work done in the MAFT project. Like FTPP, the architecture allows parallel processing within each channel (the M dimension). As in Delta-4, the focus has been on the use of COTS operating systems. The architecture also includes several completely innovative aspects: support for multiple levels of integrity, a novel error-filtering technique (α-count), and support for a wide range of scheduling models, including preemptive scheduling. At the time of writing, the first prototypes were nearing completion and it is hoped that performance measurements and fault injection results will soon be available.

A particularly constraining design requirement was that only COTS operating systems were to be used. This requirement appears to leave just two options to the fault-tolerant system designer. The first option is to use a hardware-intensive approach so that the hardware, although fault-tolerant, presents a standard interface to an unmodified COTS operating system, e.g., as in [1]. This approach, which precludes the use of COTS hardware boards, might nevertheless be necessary in some performance-critical applications. The second option, which is the one followed in GUARDS, is to use high-granularity replication managed by software above the COTS operating systems of interconnected COTS processor boards. One consequence of this choice is that the operating systems themselves cannot be protected from errors. This means that even a transient fault might require a processor to be completely reinitialized. Furthermore, the fault tolerance management software cannot access data structures that are internal to the operating systems. This leads to a nontrivial channel reintegration procedure that relies on programmer-defined context objects. One interesting direction for future research on this aspect would be to explore how compile-time reflection could be used to render context definition transparent to the application programmer [58].

ACKNOWLEDGMENTS

GUARDS is partially financed by the European Commission as ESPRIT project no. 20716. The consortium consists of three end-user companies: Technicatome (France), Ansaldo Segnalamento Ferroviario (Italy), and Matra Marconi Space (France); two technology-provider companies: Intecs Sistemi (Italy), Siemens AG Österreich PSA (Austria); and three academic partners: LAAS-CNRS (France), Pisa Dependable Computing Centre (Italy), and the University of York (United Kingdom). The University of Ulm (Germany) also participated in the first phase of the project as a subcontractor.

REFERENCES

[1] M. Abbott, D. Har, L. Herger, M. Kaufmann, K. Mak, J. Murdock, C. Schulz, T.B. Smith, B. Tremaine, D. Yeh, and L. Wong, "Durable Memory RS/6000™ System Design," *Proc. 24th Int'l Conf. Fault-Tolerant Computing (FTCS-24),* pp. 414-423, Austin, Texas, June 1994.

[2] T.A. Anderson and P.A. Lee, *Fault Tolerance—Principles and Practice.* Prentice Hall, 1981. (see also: P.A. Lee, and T. Anderson, *Fault Tolerance—Principles and Practice, Dependable Computing and Fault-Tolerant Systems,* vol. 3. Vienna: Springer-Verlag, 1990.

[3] J. Arlat, "Preliminary Definition of the GUARDS Validation Strategy," Research Report no. 96378, LAAS-CNRS, Toulouse, France, Jan. 1997. Also available as ESPRIT Project 20716 GUARDS Report no. D3A1.A0.5002.C.

[4] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation—A Methodology and Some Applications," *IEEE Trans. Software Eng.,* vol. 16, no. 2, pp. 166-182, Feb. 1990.

[5] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell, "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems," *IEEE Trans. Computers,* vol. 42, no. 8, pp.913-923, Aug. 1993.

[6] N. Audsley, "Flexible Scheduling for Hard Real-Time Systems," DPhil thesis, Dept. of Computer Science, Univ. of York, UK, 1993.

[7] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings, "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling," *Software Eng. J.,* vol. 8, no. 5, pp. 284-292, 1993.

[8] N. Audsley, K. Tindell, and A. Burns, "The End of the Line for Static Cyclic Scheduling?" *Proc. Fifth Euromicro Workshop Real-Time Systems,* pp. 36-41, Oulu, Finland, 1993.

[9] P. Barrett, A. Burns, and A.J. Wellings, "Models of Replication for Safety Critical Hard Real-Time Systems," *Proc. 20th IFAC/IFIP Workshop Real-Time Programming (WRTP '95),* pp. 181-188, Ft. Lauderdale, Fla., Nov. 1995.

[10] I. Bates and A. Burns, "Schedulability Analysis of Fixed Priority Real-Time Systems with Offsets," *Proc. Ninth Euromicro Workshop on Real-Time Systems,* pp. 153-160, Toledo, Spain, 1997.

[11] C. Béounes, M. Aguera, J. Arlat, S. Bachmann, C. Bourdeau, J.-E. Doucet, K. Kanoun, J.-C. Laprie, S. Metge, J. Moreira de Souza, D. Powell, and P. Spiesser, "SURF-2: A Program for Dependability Evaluation of Complex Hardware and Software Systems," *Proc. 23rd Int'l Conf. Fault-Tolerant Computing (FTCS-23),* pp. 668-673, Toulouse, France, June 1993.

[12] C. Bernardeschi, A. Fantechi, S. Gnesi, and A. Santone, "Formal Specification and Verification of the Inter-Channel Consistency Network," PDCC, Pisa, Italy, ESPRIT Project 20716 GUARDS Report no. I3A4.AO.6009.B, Apr. 1998.

[13] C. Bernardeschi, A. Fantechi, S. Gnesi, and A. Santone, "Formal Specification and Verification of the Inter-Channel Fault Treatment Mechanism," PDCC, Pisa, Italy, ESPRIT Project 20716 GUARDS Report no. I3A4.AO.6013.A, May 1998.

[14] C. Bernardeschi, A. Fantechi, S. Gnesi, and A. Santone, "Formal Validation of Fault Tolerance Mechanisms," *Digest of FastAbstracts —28th Fault-Tolerant Computing Symp. (FTCS-28),* pp. 66-67, Munich, Germany, 23-25 June 1998.

[15] K.J. Biba, "Integrity Considerations for Secure Computer Systems," The Mitre Corporation, Technical Report no. MTR-3153, Rev. 1, Apr. 1977.

[16] D.M. Blough, G.F. Sullivan, and G.M. Mason, "Intermittent Fault Diagnosis in Multiprocessor Systems," *IEEE Trans. Computers,* vol. 41, no. 11, pp. 1,430-1,441, Nov. 1992.

[17] A. Bondavalli, S. Chiaradonna, F.D. Giandomenico, and F. Grandoni, "Discriminating Fault Rate and Persistency to Improve Fault Treatment," *Proc. 27th Int'l Symp. Fault-Tolerant Computing (FTCS-27),* pp. 354-362, Seattle, Wash., June 1997.

[18] A. Bondavalli, S. Chiaradonna, F.D. Giandomenico, and F. Grandoni, "Inter-Channel State Restoration," PDCC, Pisa, technical note, Nov. 1997. Also available as ESPRIT Project 20716 GUARDS Report no. I1-SA4.TN.6006.B.

[19] A. Bondavalli, F.D. Giandomenico, F. Grandoni, D. Powell, and C. Rabéjac, "State Restoration in a COTS-based N-Modular Architecture," *Proc. First Int'l Symp. Object-Oriented Real-Time Distributed Computing (ISORC'98),* pp. 174-183, Kyoto, Japan, 20-22 Apr. 1998.

[20] A. Bondavalli, I. Mura, and M. Nelli, "Analytical Modelling and Evaluation of Phased-mission Systems for Space Applications," *Proc. Second Workshop High Assurance Systems Eng. (HASE-97),* Washington D.C., Aug. 1997.

[21] A. Bouali, S. Gnesi, and S. Larosa, "The Integration Project for the JACK Environment," *Bulletin of the EATCS,* vol. 54, pp.207-223, Oct. 1994. See also http://rep1.iei.pi.cnr.it/projects/JACK.

[22] A. Burns, N. Hayes, and M.F. Richardson, "Generating Feasible Cyclic Schedules," *Control Eng. Practice,* vol. 3, no. 2, pp. 151-162, 1995.

[23] A. Burns and A. Wellings, "HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems," *Real-Time Safety Critical Systems,* vol. 3, p. 313. Elsevier 1995.

[24] J. Carreira, H. Madeira, and J.G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers," *IEEE Trans. Software Eng.,* vol. 24, no. 2, pp. 125-136, Feb. 1998.

[25] Y. Dutuit, E. Châtelet, J.-P. Signoret, and P. Thomas, "Dependability Modelling and Evaluation by Using Stochastic Petri Nets: Application to Two Test Cases," *Reliability Eng. & System Safety,* vol. 55, pp. 117-124, 1997.

[26] A. Fantechi, S. Gnesi, F. Mazzanti, R. Pugliese, and E. Tronci, "A Symbolic Model Checker for ACTL," PDCC, Pisa, Italy, ESPRIT Project 20716 GUARDS Report no. I3A5.AO.6011.A, Apr. 1998.

[27] A. Fantechi, S. Gnesi, and L. Semini, “Formal Description and Validation for an Integrity Policy Supporing Multiple Levels of Criticality,” *Proc. Seventh IFIP Working Conf. Dependable Computing for Critical Applications (DCCA-7),* pp. 6-8, San Jose, Calif., Jan. 1999.

[28] L. Gong, P. Lincoln, and J. Rushby, “Byzantine Agreement with Authentication: Observations and Applications in Tolerating Hybrid and Link Faults,” *Dependable Computing for Critical Applications 6, Dependable Computing and Fault-Tolerant Systems,* R.K. Iyer, M. Morganti, W.K. Fuchs and V. Gligor, eds., vol. 10, pp. 139-157, 1998.

[29] J. Gray, “Why Do Computers Stop and What Can Be Done about It?” *Proc. Fifth Symp. Reliability in Distributed Software and Database Systems,* pp. 3-12, Los Angeles, Jan. 1986.

[30] R.E. Harper and J.H. Lala, “Fault-Tolerant Parallel Processor,” *J. Guidance, Control and Dynamics,* vol. 14, no. 3, pp.554-563, May-June 1990.

[31] *HOOD Reference Manual, Release 3.1.1,* HOOD Technical Group 1992.

[32] M.-C. Hsueh, T.K. Tsai, and R.K. Iyer, “Fault Injection Techniques and Tools,” *Computer,* vol. 40, no. 4, pp. 75-82, Apr. 1997.

[33] “Functional Safety: Safety-Related Systems,” *Draft Int'l Standard IEC 1508,* Int'l Electrotechnical Commission, IEC Document no. 65A/179/CDV, Geneva, June 1995.

[34] “HRT-HoodNICE: a Hard Real-Time Software Design Support Tool,” Intecs Sistemi, Pisa, Italy, ESTEC Contract 11234/NL/FM(SC), Final Report, 1996.

[35] E. Jenn, “Modelling for Evaluation,” Technicatome, Aix en Provence, France, ESPRIT Project 20716 GUARDS Report no. I3A3.TN.0056.A Jan. 1998.

[36] K. Kanoun, M. Borrel, T. Morteveille, and A. Peytavin, “Modelling the Dependability of CAUTRA, a Subset of the French Air Traffic Control System,” *Proc. 26th Fault-Tolerant Computing Symp. (FTCS-26),* pp. 106-115, Sendai, Japan, June 1996.

[37] R.M. Kieckhafer, C.J. Walter, A.M. Finn, and P.M. Thambidurai, “The MAFT Architecture for Distributed Fault Tolerance,” *IEEE Trans. Computers,* vol. 37, no. 4, pp. 398-405, Apr. 1988.

[38] N. Krishnamurthy, V. Jhaveri, and J.A. Abraham, “A Design Methodology for Software Fault Injection in Embedded Systems,” *Proc. IFIP Int'l Workshop Dependable Computing and Its Applications (DCIA-98),* Y. Chen, ed., pp. 237-248, Johannesburg, South Africa, Jan. 1998.

[39] J.H. Lala and R.E. Harper, “Architectural Principles for Safety-Critical Real-Time Applications,” *Proc. IEEE,* vol. 82, no. 1, pp. 25-40, Jan. 1994.

[40] L. Lamport and P.M. Melliar-Smith, “Synchronizing Clocks in the Presence of Faults,” *J. ACM,* vol. 32, no. 1, pp. 52-78, Jan. 1985.

[41] “Dependability: Basic Concepts and Terminology,” *Dependable Computing and Fault-Tolerance,*J.-C. Laprie, ed., vol. 5, p. 265. Vienna: Springer-Verlag, 1992.

[42] J.-C. Laprie, J. Arlat, J.-P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J.-C. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, C. Mazet, D. Powell, C. Rabéjac, and P. Thévenod, *Dependability Handbook,* p. 324, Toulouse, France: Cépaduès-Editions, 1995. (in French; English version in preparation).

[43] S. Lee and K.G. Shin, “Optimal Multiple Syndrome Probabilistic Diagnosis,” *Proc. 20th Int'l Symp. Fault-Tolerant Computing Systems (FTCS-20),* pp. 324-31, Newcastle upon Tyne, U.K., 1990.

[44] J. Leung and J. Whitehead, “On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks,” *Performance Evaluation,* vol. 2, no. 4, pp. 237-250, 1982.

[45] P. Lincoln and J. Rushby, “A Formally Verified Algorithm for Interactive Consistency Under a Hybrid Fault Model,” *Proc. 23rd Int'l Conf. Fault-Tolerant Computing (FTCS-23),* pp. 402-411, Toulouse, France, 1993.

[46] J. Lundelius-Welch and N. Lynch, “A New Fault-Tolerant Algorithm for Clock Synchronization,” *Information & Computation,* vol. 77, no. 1, pp. 1-16, 1988.

[47] P.M. Melliar-Smith and R.L. Schwartz, “Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight Control System,” *IEEE Trans. Computers,* vol. 31, no. 7, pp. 616-630, July 1982.

[48] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas, “PVS: Combining Specification, Proof Checking, and Model Checking,” *Computer-Aided Verification (Proc. CAV'96),* New Brunswick, N.J., July/Aug. 1996.

[49] A. Paganone and P. Coppola, “Specification and Preliminary Design of the Architectural Development Environment,” Intecs Sistemi, Pisa, Italy, ESPRIT Project 20716 GUARDS Report no. D2A1.A0.3002.C, Apr. 1997.

[50] M. Pease, R. Shostak, and L. Lamport, “Reaching Agreement in the Presence of Faults,” *J. ACM,* vol. 27, no. 2, pp. 228-234, Apr. 1980.

[51] S. Poledna, “Deterministic Operation in of Dissimilar Replicated Tasks Sets in Fault-Tolerant Distributed Real-Time Systems,” *Dependable Computing for Critical Applications 6,* M. Dal Cin, C. Meadows, and W.H. Sanders, eds., pp. 103-119, 1998.

[52] D. Powell, “Distributed Fault-Tolerance—Lessons from Delta-4,” *IEEE Micro,* vol. 14, no. 1, pp. 36-47, Feb. 1994.

[53] D. Powell, “Preliminary Definition of the GUARDS Architecture,” LAAS-CNRS, Toulouse, France, Research Report no. 96277, Jan. 1997. Also available as ESPRIT Project 20716 GUARDS Report no. D1A1.A0.5000.D.

[54] D. Powell, J. Arlat, and K. Kanoun, “Generic Architecture Instantiation Guidelines,” LAAS-CNRS, Toulouse, France, Research Report no. 98136, May 1998. Also available as ESPRIT Project 20716 GUARDS Report no. I1SA1.TN.5008.C.

[55] D. Powell, C. Rabéjac, and A. Bondavalli, “Alpha-Count Mechanism and Inter-Channel Diagnosis,” ESPRIT Project 20716 GUARDS Report no. I1SA1.TN.5009.E, 1998.

[56] C. Rabéjac, “Inter-Channel Fault Treatment Mechanism,” Matra Marconi Space, France, Guards Report no. D1A3 AO 2014 B, Mar. 1997.

[57] P. Ramanathan, K.G. Shin, and R.W. Butler, “Fault-Tolerant Clock Synchronization in Distributed Systems,” *Computer,* pp. 33-42, Oct. 1990.

[58] J.C. Ruiz Garcia, M.-O. Killijian, J.-C. Fabre, and S. Chiba, “Optimized Object State Checkpointing Using Compile-Time Reflection,” *Workshop Embedded Fault-Tolerant Systems,* pp. 46-48, Boston, 1998.

[59] W.H. Sanders and W.D. Obal II, “Dependability Evaluation Using UltraSAN,” *Proc. 23rd Int'l Conf. Fault-Tolerant Computing (FTCS-23),* pp. 674-679, Toulouse, France, 1993.

[60] D. Schwier and F. von Henke, “Mechanical Verification of Clock Synchronization Algorithms,” Design for Validation, ESPRIT Long Term Research Project 20072: DeVa - Second Year Report, LAAS-CNRS, Toulouse, France, pp. 287-303, 1997.

[61] L. Semini, “Formal Specification and Verification for an Integrity Policy Supporting Multiple Levels of Criticality,” PDCC, Pisa, Italy, ESPRIT Project 20716 GUARDS Report, no. I3A5.AO.6012.A, Apr. 1998.

[62] H. Simpson, “Four-Slot Fully Asynchronous Communication Mechanism,” *IEE Proc,* vol. 137, no. Py. E 1, pp. 17-30, Jan. 1990.

[63] T.K. Srikanth and S. Toueg, “Optimal Clock Synchronization,” *J. ACM,* vol. 34, no. 3, pp. 626-645, July 1987.

[64] P. Thambidurai and Y.-K. Park, “Interactive Consistency with Multiple Failure Modes,” *Proc. Seventh Symp. Reliable Distributed Systems (SRDS-7),* pp. 93-100, Columbus, Ohio, 1988.

[65] K. Tindell, “Fixed Priority Scheduling of Hard Real-Time Systems,” DPhil thesis, Dept. of Computer Science, Univ. of York, U.K., 1993.

[66] E. Totel, L. Beus-Dukic, J.-P. Blanquart, Y. Deswarte, D. Powell, and A. Wellings, “Integrity Management in GUARDS,” *Proc. IFIP Int'l Conf. Distributed Systems Platforms and Open Distributed Processing (Middleware '98),* pp. 105-122, The Lake District, England, 15-18 Sept. 1998.

[67] E. Totel, J.-P. Blanquart, Y. Deswarte, and D. Powell, “Supporting Multiple Levels of Criticality,” *Proc. 28th Int. Symp. Fault-Tolerant Computing (FTCS-28),* pp. 70-79, Munich, Germany, 23-25 June 1998.

[68] A. Wellings and L. Beus-Dukic, “Guidelines for Mapping HRT-HOOD to POSIX/C,” Univ. of York, U.K., ESPRIT Project 20716 GUARDS Report no. I2A1-2.A0.7041.B, Dec. 1997.

[69] A. Wellings, L. Beus-Dukic, and D. Powell, “Real-Time Scheduling in a Generic Fault-Tolerant Architecture,” *Proc. 19th Real-Time Systems Symp. (RTSS-19),* pp. 390-398, Madrid, 2-4 Dec. 1998.

David Powell (M'92) received his Bachelor of Science degree in electronic engineering from the University of Southampton, England, in 1972, a Specialty Doctorate degree from the Toulouse Paul Sabatier University in 1975, and his Docteur ès-Sciences degree from the Toulouse National Polytechnic Institute in 1981. He has been at LAAS-CNRS since 1972, where he holds a position of "Directeur de Recherche CNRS" and leads the Dependable Computing and Fault Tolerance Research Group.

He has managed several national and European research contracts and acted as a consultant for several aerospace, telecommunication, and data processing companies in France and for the European Commission. He was the scientific director of the six year Delta-4 Esprit project on open dependable distributed computing and the Scientific Advisor of the GUARDS Esprit project, described in this paper.

Dr. Powell's research work concerns the design and validation of fault-tolerant distributed computing systems. He has written more than 90 papers for international and national journals and conferences, is coauthor of two books on dependable computing, and holds a patent for a fault- and damage-tolerant network for data transmission. Dr. Powell is a member of the ACM and the IEEE, the IFIP 10.4 working group on Dependable Computing and Fault Tolerance, and the SEE Dependable Computing Club.

Jean Arlat (M'80) received the Engineer degree from the National Institute of Applied Sciences of Toulouse in 1976, and the Doctor in Engineering degree and the Docteur ès-Sciences degree from the National Polytechnic Institute of Toulouse in 1979 and 1990, respectively. He is currently "Directeur de Recherche" within CNRS, the French National Organization for Scientific Research and a member of the group on Dependable Computing and Fault Tolerance at LAAS-CNRS. Since 1997, he has been director of the Laboratory for Dependable Engineering (LIS: Laboratoire d'Ingénierie de la Sûreté de fonctionnement) hosted by LAAS.

Dr. Arlat's research interests focus on the evaluation of hardware and software fault-tolerant systems, including both analytical modeling and experimental fault injection approaches, subjects on which he authored or coauthored more than 70 papers for international and national journals and conferences.

Dr. Arlat chaired the IEEE Computer Society's Technical Committee on Fault-Tolerant Computing (TC-FTC) in 1994-1995. Since January 1999, he has been chairman of the IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance. He is a member of the ACM, the IEEE, and the SEE Dependable Computing Club.

Ljerka Beus-Dukic received the BSc and MSc degrees in electrical engineering/computer science from the Faculty of Electrical Engineering, University of Sarajevo, Bosnia and Herzegovina, in 1977 and 1988, respectively, and the DPhil degree in computer science from the University of York, England, in 1992. From 1977 to 1988, and from 1991 to 1996, she worked in industry as a software engineer on supervisory control and data acquisition software for industrial real-time applications. From 1996 to 1999, she was a research fellow in the Department of Computer Science, University of York, England.

Dr. Beus-Dukic's research interests include real-time operating systems, real-time communication, safety-critical systems, requirements engineering, reuse, and acquisition of COTS software components. She is currently a senior lecturer at the University of Northumbria at Newcastle.

Andrea Bondavalli (M'97) received his "Laurea" degree in computer science from the University of Pisa in 1986. In the same year, he joined CNUCE-CNR, where he holds a position of "ricercatore" working in the Dependable Computing System group. From April 1991 to February 1992, he was a guest member of the staff at the Computing Laboratory of the University of Newcastle-upon-Tyne (UK).

Since 1989, he has been working on several European projects, such as Esprit BRA PDCS, Esprit BRA PDCS2, Esprit 27439 HIDE, and various national projects on dependable computing He is currently managing the PDCC partnership in the Esprit 20716 GUARDS project. He also acted as a consultant for transportation companies and software houses.

Dr. Bondavalli has served as a program committee member for international conferences and symposia and as a reviewer for conferences and journals. His current research interests include the design of dependable real-time computing systems, software and system fault tolerance, the integration of fault tolerance in real-time systems, and the modeling and evaluation of dependability attributes like reliability, availability, and performability. Dr. Bondavalli is a member of the IEEE Computer Society and the AICA Working Group on "Dependability in Computer Systems."

Paolo Coppola received his university degree in computer science in 1980 from the University of Pisa. He worked at CSELT (a research laboratory of the Italian telephone company) from 1981 to 1984. In 1984, he joined INTECS, where he currently leads the "Embedded Space Systems" group.

He has managed several national and European development projects and research contracts, mainly concerning embedded software systems for space, defense, and industrial applications. Particular interests are design methods for hard-real-time systems, including techniques and tools for the specification and verification of timing requirements and constraints.

Dr. Coppola is member of the EUROSPACE Software Engineering panel and the program committee of the "Data Systems in Aerospace" (DASIA) conference.

Alessandro Fantechi received is "Laurea" degree in computer science from the University of Pisa in 1978. He has been an associate professor on the Faculty of Engineering of the University of Florence since 1995, and an associate of the Istituto di Elaborazione dell'Informazione (IEI) del CNR, Pisa since 1992. Previously, he was an associate professor at the University of Pisa (from 1992) and a researcher at IEI-CNR (from 1983). He was involved in the Esprit LOTOSPHERE project before the current involvement in the GUARDS project described in this paper. He has taught foundations of computer science, software engineering, and dependable systems engineering at the Universities of Florence and Pisa, and presented several software engineering lectures at main Italian aerospace industries.

Prof. Fantechi's current research is focused on the industrial application of formal verification methods. In this field, he has recently worked for a pilot project with Ansaldo Trasporti on the verification of a railway signaling system. Other research interests have included formal description techniques, temporal logic, distributed systems programming and modeling, and applications of natural language understanding to requirements engineering. He has written more than 40 papers for international journals and conferences. Prof. Fantechi is a member of the ACM.

Eric Jenn received his Engineer Degree in computer science from the National Institute of Applied Sciences, Toulouse, France, in 1989 and a PhD in computer science from the Toulouse National Polytechnic Institute in 1994.

His main interests include hardware and software fault tolerance for real-time dependable systems, validation by modeling and fault injection, and object-oriented design.

He is currently working as a software engineer at Technicatome, the company which develops the nuclear reactors of the submarines and the nuclear aircraft carrier of the French navy.

Christophe Rabéjac received a degree from the French Ecole Polytechnique (Palaiseau) in 1991, a degree from ENSEEIHT (National School for Engineers in Computing Science, Toulouse) in 1993, and his PhD from the Toulouse National Polytechnic Institute in 1995. He has been at Matra Marconi Space France (MMS) since 1992.

He has been involved in the architecture of a distributed mission-planning software tool for ground segments, implemented above COTS CORBA technology. He is currently the MMS manager of the GUARDS Esprit Project and, in that context, is in charge of technological developments in fault-tolerant architectures for MMS.

Dr. Rabéjac's main fields of interest are avionics architectures for real-time critical embedded systems and all related software techniques. He has published several papers for international conferences such as FTCS and Safecomp.

Andy Wellings is a professor of real-time systems at the University of York, UK in the Computer Science Department. He is interested in most aspects of the design and implementation of real-time dependable computer systems. He has authored/coauthored more than 150 papers/reports. He is European editor-in-chief for the computer science journal *Software-Practice and Experience*.

Professor Wellings teaches courses in operating systems, real-time systems, and network and distributed systems. He is the author of several books, including "Hard Real-Time HOOD: A Structured Design Method for Hard Real-Time Ada Systems," "Concurrency in Ada," and "Real-Time Systems and Programming Languages."