Fault-Tolerant Distributed Deployment of Embedded Control Software

Claudio Pinello, Luca P. Carloni, *Member, IEEE*, and Alberto L. Sangiovanni-Vincentelli, *Fellow, IEEE*

***Abstract*—Safety-critical feedback-control applications may suffer faults in the controlled plant as well as in the execution platform, i.e., the controller. Control theorists design the control laws to be robust with respect to the former kind of faults while assuming an idealized scenario for the latter. The execution platforms supporting modern real-time embedded systems, however, are distributed architectures made of heterogeneous components that may incur transient or permanent faults. Making the platform fault tolerant involves the introduction of design redundancy with obvious impact on the final cost. We present a design flow that enables the efficient exploration of redundancy/cost tradeoffs. After providing a system-level specification of the target platform and the fault model, designers can rely on the synthesis of the low-level fault-tolerance mechanisms. This is performed automatically as part of the embedded software deployment through the combination of the following three steps: replication, mapping, and scheduling. Our approach has a sound foundation in fault-tolerant data flow, a novel model of computation that simplifies the integration of formal validation techniques. Finally, we report on the application of our design flow to two case studies from the automotive industry: a steer-by-wire system from General Motors and a drive-by-wire system from BMW.**

***Index Terms*—Automotive electronics, embedded control software, fault tolerance, real-time embedded systems.**

I. Introduction

EMBEDDED software has a pervasive presence in our world, from a variety of consumer electronic products to many safety-critical applications in industries such as manufacturing, health, aerospace, and automotive. Increasingly, embedded software is taking over the role of mechanical and dedicated electronic systems in engaging the physical world [1]. As more than 98% of the 8.2-billion microprocessor/microcontroller units shipped in 2000 were related to embedded applications [2], embedded computing is becoming a key source of innovation in engineered systems [3]. For instance, more than 90% of the innovation (and hence value added) in a car is in electronics, and electronic components comprise more than 30% of a car's manufacturing cost [4]. As the portion of the electronic systems' development cost that is related to embedded software programming continues to grow, the distributed nature of many new important classes of embedded applications adds a new level of design complexity requiring the deployment of tightly interactive concurrent processes on distributed (networked) execution platforms.

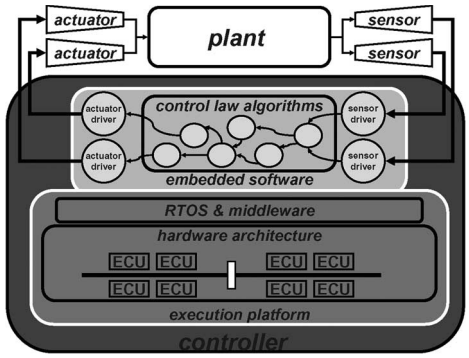


Fig. 1. General structure of a real-time control system.

The general structure of a real-time feedback-control system is shown in Fig. 1: The controller is made of embedded software that implements a set of control-law algorithms, runs on a hardware execution platform, and interacts with the plant by means of sensors and actuators. An execution platform is a heterogeneous system that is typically made of an infrastructure-software layer (real-time operating systems, middleware, and drivers) on top of an underlying hardware layer (a set of processing elements, called the electronic control units or ECUs,[1] storage elements, and communication channels like buses or crossbars). The hardware layer often has a distributed nature that is inherited from the physical characteristics of the plant. The design of an embedded system requires the definition of the execution platform, through the selection and assembly of its components, and the programming of the control software. This consists of a set of concurrent processes (sometimes called tasks in this paper), which implement the control law and whose execution on the target platform must meet hard real-time constraints [5]. This complex engineering task is more challenging in the case of safety-critical applications, e.g., drive-by-wire in automobiles [6], due to the requirement of making the design resilient to faults.

Technically, a fault is the cause of an error, an error is the part of the system state that may cause a failure, and a failure is the deviation of the system from its specification [7]. A deviation from the specification may be due to accidents

Manuscript received March 21, 2007; revised July 23, 2007. This work was supported in part by BMW, by General Motors, by the National Science Foundation under Award 0644202, and by the Center for Hybrid and Embedded Software Systems (CHESS), which is funded by the National Science Foundation under Award CCF-0424422. This paper was recommended by Associate Editor Y. Paek.

C. Pinello was with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720 USA. He is now with Cadence Research Laboratories, Berkeley, CA 94704 USA (e-mail: ClaudioPinello@Cal.Berkeley.edu).

L. P. Carloni is with the Department of Computer Science, Columbia University, New York, NY 10027 USA (e-mail: luca@cs.columbia.edu).

A. L. Sangiovanni-Vincentelli is with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720 USA (e-mail: alberto@eecs.berkeley.edu).

Digital Object Identifier 10.1109/TCAD.2008.917971

[1]The term ECU is widely used in automotive systems to indicate (packaged) boards/computing nodes. Our work may be relevant to other domains as well.

occurring during system operations (faults) or designers' mistakes (bugs). Theoretically, all bugs can be eliminated before the system is deployed, and practically, they are minimized by using design environments that are based on sound models of computation (MoCs) like the synchronous paradigm [8]. These have well-defined semantics that enable the application of formal validation techniques [9]–[11]. On the other hand, faults must be addressed online as the system is operating.

We classify faults in two categories that are relevant for feedback-control systems: plant faults and execution platform faults. Plant faults, including faults in sensors and actuators, must be handled at the algorithmic level by control theorists who know the stability requirements of the plant and its controllable/uncontrollable modes and can employ estimation techniques and adaptive control methods. For instance, the control of a drive-by-wire system might require handling properly a tire puncture or the loss of one of the four brakes.

Faults in the execution platform may affect the infrastructure-software layer or the underlying hardware layer. For instance, a loss of power may turn off an ECU momentarily or forever. Making the controller fault tolerant involves the introduction of redundancy in the design by replicating platform components as well as embedded software processes. Redundancy has an obvious impact on costs: While choosing a faster microprocessor, duplicating a bus, or replicating some embedded software may not affect sensibly the overall cost of a new airplane, their impact may be striking for high-volume products like the ones of the automotive industry.

The analysis of the tradeoffs between higher redundancy and lower costs is a challenging hardware–software codesign effort that designers of fault-tolerant systems for cost-sensitive applications must face aside from making decisions on the following points: 1) how to introduce redundancy and 2) how to deploy the redundant design on a distributed execution platform. Because these two activities are both tedious and error prone, designers often rely on off-the-shelf solutions to address fault tolerance, like time-triggered architectures [12]. These allow the application to be unaware of fault-tolerant mechanisms, which are transparently provided by the architecture to cover the platform faults. Thus, designers may focus on avoiding design bugs and tuning the control algorithms to address the plant faults. However, the rigidity of off-the-shelf solutions may lead to suboptimal results from a design cost viewpoint.[2]

A. Contributions and Paper Organization

We present a design methodology for safety-critical embedded control applications and a companion design flow called Safety-Critical Real-time APplications Exploration (SCRAPE) [13], [14]. This interactive software environment assists designers with the exploration of the redundancy/cost tradeoffs and the derivation of the final fault-tolerant implementation. In Section II, we give an overview of SCRAPE with the help of a paradigmatic feedback-control application, the inverted-pendulum control system. SCRAPE has a formal foundation on fault-tolerant data flow (FTDF), a novel synchronous and deterministic MoC that we describe in Section III. Determinism and synchrony simplify the integration of formal validation techniques in the design flow. Our approach is based on the principle of separation of concerns. Designers start specifying the application functionality without committing to a specific execution platform and, therefore, independently from the faults that it may suffer.[3] This decoupling enables reuse of the control algorithms on new versions of the product or different products. At later stages, designers define a possible architecture for the target execution platform, identify the expected set of platform faults (fault model), and annotate the embedded software processes to express their relative criticality. While the FTDF model is fault-model independent, currently, SCRAPE supports only two fault models: fail-silent execution platforms and platforms that produce detectably faulty results. In Sections III and V-A, we discuss how to extend them to random errors and Byzantine faults.

The algorithm specification, the process criticality, the platform architecture, and the fault model are processed simultaneously by SCRAPE in order to achieve the following.

1) Automatically deduce the necessary software replication.
2) Distribute each process on the execution platform.
3) Derive an optimal scheduling of the processes on each ECU to satisfy the overall timing constraints.

When combined, the three steps (replication, mapping, and scheduling) give the automatic fault-tolerant deployment of the embedded software on the distributed execution platform (Section IV). The deployment is robust with respect to both permanent and transient faults in the execution platform. A final validation step checks if the result satisfies the timing constraints for the control application (Section V). If this is not the case, precise guidelines are returned to the designers who may use them to refine the control algorithms, upgrade (and increase the redundancy of) the platform, and/or revisit the fault model. In Section VI, we apply SCRAPE to two case studies developed in collaboration with two major automotive companies.

B. Related Work

While there is an extensive literature on fault tolerance for distributed systems [15], [16], our approach focuses on embedded control applications, and it is closer to the works by Izosimov *et al.* [17]–[19]. They address the problem of transient faults in the processing elements, assuming a model that combines time-triggered communication protocol [20] and cyclic static scheduling of the processors. The use of static schedules, particularly with transparent recovery [21], can provide high determinism and simplify debugging, typically at the expense of performance and schedulability. In [17], reexecution (time redundancy) and replication (space redundancy) are optimized automatically to improve schedulability. In [19], they refine this approach to include checkpointing, thus reexecuting only the parts of a process that were affected by transient faults, rather than the entire process. A method to explore the tradeoff between higher schedulability and higher transparency, using only reexecution, is proposed in [18].

Our approach differs in a number of aspects. First, we tackle primarily permanent faults (using replication only) on

[2]Although centered on a synthesis step, our approach does not exclude the use of predesigned components like TTA modules, protocols like TTP, or fault-tolerant operating systems. These can be part of a library of building blocks that designers use to further explore the fault-coverage/cost tradeoff.

[3]Only the sensor/actuator set is defined so that control laws can be devised.

both processors and channels. We check for faults at each communication and mask faults by space redundancy. This gives coverage against transient faults as well, typically with higher overhead than time redundancy. We consider more general architecture topologies, whereas they consider a single (fault-tolerant) broadcast bus, based on TTP. Our model is more abstract and can be applied to both event and time-triggered protocols [22]. It models mutual exclusion but does not address specific protocol constraints, like time slots reuse or priority-based arbitration. In fact, we build on the work in [23] extending the model in different ways and implementing automation of the methods. First, we introduced FTDF, a MoC that specifically targets real-time control systems where sensors and actuators have distinct roles and cannot be duplicated automatically. In particular, in [23], the failure of a sensor even if replicated cannot be tolerated because the data-flow subset depending on that sensor could not be executed. In FTDF, instead, different processes can have different firing rules and can execute in spite of the failure of some sensors. Moreover, processes can have different criticalities and require more or less fault tolerance, and degraded modes of operation can be modeled. In SCRAPE, designers can select the scheduling policy between the following: 1) time-triggered schedules (providing transparency properties as in [17] and [21]) and 2) dynamic schedules (where processes are activated by data arrival or aborted by watchdogs, thus resulting in higher schedulability). Finally, we extended the synthesis algorithm to cope with replica determinism.

II. Proposed Methodology and Design Flow

The SCRAPE design flow covers various phases from the conception of the control algorithms to the validation of the automatic deployment on the target execution platform. It is organized in the following six main stages: 1) definition of the control strategy; 2) identification of process criticality; 3) specification of the execution platform and its fault model; 4) specification of the fault behavior; 5) fault-tolerant embedded software deployment; and 6) validation of the system implementation.

The first four stages are interactive because it is essential to rely on the experience of the designers and their knowledge of the specific features of the given control application. However, the fault-tolerant deployment of embedded software and the validation of the final system implementation are error-prone tasks, whose complexity grows dramatically with the problem size. Hence, these stages are fully automatic, as detailed in Sections IV and V. Fig. 2 shows the main stages of SCRAPE. In the rest of this section, we describe each stage with the help of a simple example that we will use throughout this paper.

Running Example: Inverted-Pendulum Control System. The plant is the inverted pendulum, and the controller attempts to keep it around the vertical unstable position. The pendulum controller at the top-left of Fig. 2 is specified according to FTDF, which is a synchronous monoperiodic MoC. Each periodic execution is a three-phase reaction, whose phases are described as follows.

1) Sampling: The input actor acquires from the (redundant) sensors separate measures of the pendulum position and assesses the position reliably through "sensor fusion."

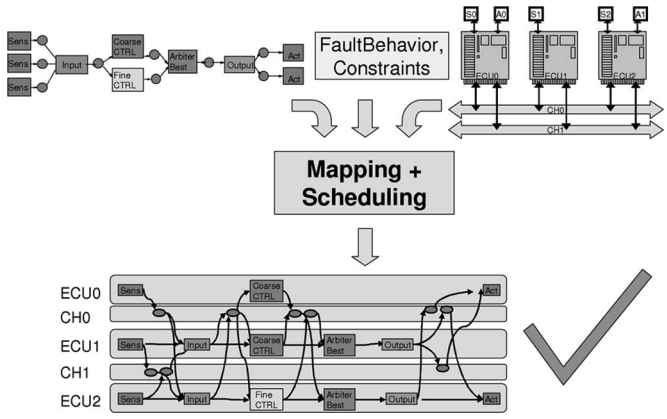


Fig. 2. SCRAPE interactive design flow.

2) Controlling: Controller actors execute the software processes implementing two types of control laws (coarse and fine) and arbitrate the value to be actuated.
3) Actuating: The output actor balances the actuation effort among the (redundant) actuators that issue commands to the electronics (A/D converters or pulsewidth-modulation drivers) to set the motor torque.

The actuator updates are applied to the plant at the period end to reduce jitter, a well-known technique in real-time control [24], [25].

A. Definition of the Control Strategy

This initial design stage is mainly in the field of expertise of the control theorists who define the control goals, identify the type and number of sensors and actuators, and specify the control algorithms and their period $T_{\max}$. The algorithms deal with faults in the plant, the sensors, and the actuators [26]. There is yet no specific assumption on the type of execution platform or on the type of faults that it may exhibit.

Running Example. The FTDF at the top-left of Fig. 2 uses three sensors to determine the position (angle) of the inverted pendulum. The actuation occurs through a pair of motors that apply their torque on a common shaft, around which the pendulum revolves. The following two control laws execute every 300 time units to bring the measured position error to "zero."

1) A coarse (bang-bang) controller: It applies maximum positive or negative torque based on the sign of the error (i.e., whether the pendulum is to the left or right of the desired position).
2) A fine (linear) controller: It applies a torque proportional to the position error.

B. Identification of Process Criticality

Designers identify different levels of criticality for the various processes. For example, essential processes provide highly safety-critical functionality, advanced processes provide safety-critical functionality with higher performance, and optional processes provide nonsafety-critical functionality.

Advanced and essential processes may express some redundancy by sharing common functionality. Then, in the absence of faults, they may both contribute to the actuated values. Arbitration processes combine such redundant values. Typically, an arbiter is as critical as the most critical of its input processes.

Running Example. The bang-bang controller is assigned higher criticality due to its robustness to actuator variability, However, the fine controller yields better performance with reduced chattering. Hence, the "arbiter" selects the fine-controller result whenever it is available, and it defaults to the coarser result otherwise. In this case, the "arbiter" process is as critical as the coarse-controller process.

C. Specification of the Execution Platform and Its Fault Model

Designers use execution platforms with two types of components: ECUs and channels (see Fig. 2, top right). We model this as a platform graph $\mathrm{PG} = (P, C, D)$, where P is the set of ECUs, C is the set of channels, and $D \subset P \times C$ is the set of edges representing the interconnections among the components. A fault model defines the class of expected faults that should be tolerated on a given PG. The FTDF algorithm does not need to be aware of the fault model of the underlying platform.[4] A fault model specifies also the number and combination of faults to be tolerated [27]. This is done based on a statistical analysis of the various components, e.g., mean time between faults and mean time between repairs. As in [23], we use failure patterns to capture the combination of the subsystems' failures of interest. A failure pattern f is a subset of vertices of PG that may fail together during the same reaction. A set $F \subset 2^{P \cup C}$ of failure patterns identifies the fault scenarios that must be tolerated.

Running Example. Consider the platform in Fig. 2 for the inverted-pendulum controller. If we need to tolerate the failure of at most one ECU at a time, then, $F = \{\emptyset, \{\mathrm{ECU0}\}, \{\mathrm{ECU1}\}, \{\mathrm{ECU2}\}\}$, where the empty failure pattern $\emptyset$ denotes absence of faults.

D. Specification of Fault Behavior and Mapping Constraints

The fault behavior captures the system fault mitigation strategy. For each failure pattern, designers specify which subset of the functionality must be guaranteed execution based on the process criticality (fault-tolerance binding). Typically, all processes must run in the absence of faults. As faults become more severe, the execution of less critical processes can be dropped. The fault-tolerance binding dictates how the system functionality degrades as faults occur. The functional binding is a set of mapping constraints and performance estimates indicating how to map each FTDF vertex. Together, these bindings specify the requirements for a redundant deployment of the FTDF on the platform graph PG.

Running Example. The desired fault behavior is as follows:

1) execution of the entire FTDF for the empty failure pattern;
2) when an ECU fails, we can drop execution of the linear controller (and of the sensor/actuator actors mapped on the faulty ECU).

The functional binding constrains sensor and actuator processes on the ECUs connected to those I/O.

[4]The examples and the automation tools presented in this paper assume *fail silence*, i.e., components either provide correct results or do not provide any result. However, FTDF is fault-model independent: More complex models can be integrated in our approach by adding support for error detection and voting mechanism in the implementation of communication media (see Section III).

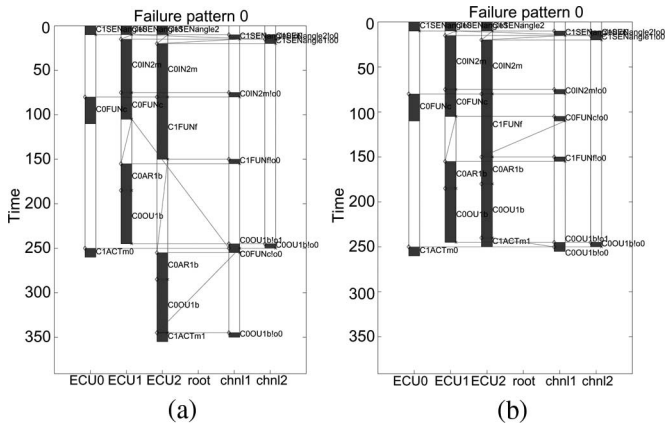


Fig. 3. Synthesized solutions for the pendulum example.

E. Fault-Tolerant Embedded Software Deployment

This stage is the core of the design automation in SCRAPE. It includes the following three steps: process replication, mapping, and scheduling. We replicate processes based on the fault model and their criticality. Assuming fail-silent execution platforms, a single replica of a process may be sufficient. For other fault modes, more replicas may be needed to compare results and prevent error propagation (see footnote 4). Then, the various replicas are mapped to resources and are scheduled. The higher the parallelism, the larger the freedom in scheduling. To drive the scheduler toward better solutions, designers can set additional precedence constraints. Other optimizations are described in the sequel. Optimal or efficient scheduling depends on accurate knowledge of the worst-case data-transmission times on the channels and the worst case execution time (WCET) of the processes [28], [29].

Running Example. The synthesized redundant deployment is shown in Fig. 3(a), where each column corresponds to a distinct resource. A series of rectangles (representing processes and messages) is assigned to a column in an order that is compatible with the data dependences specified in the FTDF (indicated by the arrows across the rectangles). The rectangle height is proportional to process/message duration. The diagram shows that the application runs at the rate of once every 355 time units. Fig. 3(b) shows an optimized solution with a reaction time of 260 time units.

F. Validation of the System Implementation

Finally, the mapped design must be validated. While there has been progress on correct-by-construction methods that complete the validation effort in the mapping phase, trace-based functional simulation remains the most common approach: Parts of the system are simulated to assess the time needed to react to a fault or the deviation of a control set point in response to a fault. Our FTDF library enables functional simulation with the injection of faults as omission errors. Formal methods for static verification assess the WCET and check whether the deadlines are met. The validation techniques supported in SCRAPE are discussed in Section V.

Running Example. Fig. 4 shows the results of the timing analysis for the different failure patterns. For example, when ECU3 fails (last diagram), none of its processes executes. Note

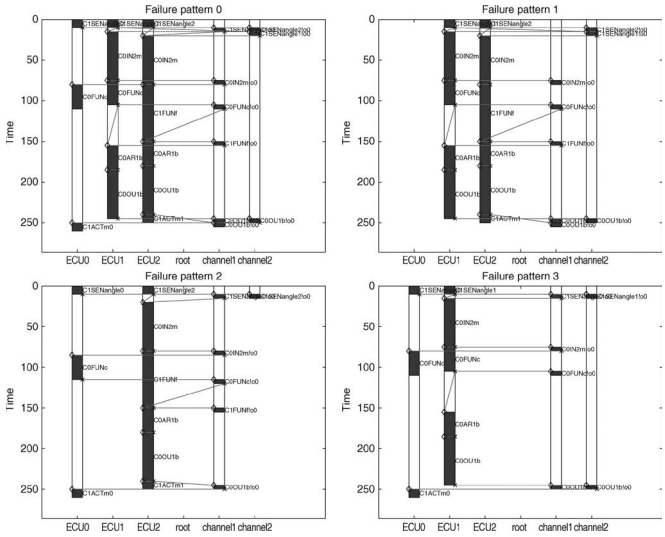


Fig. 4. Pendulum's timing analysis across the failure patterns, by rows: no faults, ECU1 fails, ECU2 fails, and ECU3 fails. No task runs on faulty ECUs.

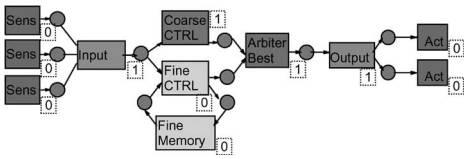


Fig. 5. Fine controller with state memory. Criticality χ is in the dotted boxes.

that, for any failure pattern in F, at least one of the two actuator actors ("C1ACTm0" and "C1ACTm1") completes the execution before 260 time units.

III. FTDF

FTDF is a MoC for specifying safety-critical feedback-control applications. It is an extension of the classic data-flow [30], [31] models. Its structure enables formal analysis and automatic/semiautomatic synthesis techniques for obtaining an efficient fault-tolerant implementation of the application under design.

A. MoC

The basic building blocks of an FTDF are actors and communication media. FTDF actors represent processes that exchange data tokens at each periodic reaction with synchronous semantics [8]. FTDF communication media provide a fault-model-independent communication semantics.

Definition 1: An FTDF is a graph $\mathcal{G} = (V, E)$ with $(V = A \cup M)$ and $E \subset (A \times M) \cup (M \times A)$, where A is the set of actors and M is the set of communication media.

$\mathcal{G}$ is bipartite, and actors are always connected via a communication medium. Each edge $e = (m, a) \in E$ (respectively $e = (a, m) \in E$) corresponds to an input (respectively output) port of actor a. Figs. 5 and 6 are examples of FTDF graphs. The set of actors is partitioned into seven sets as $A = A_S \cup A_{\mathrm{Act}} \cup A_I \cup A_O \cup A_T \cup A_A \cup A_M$, corresponding to six regular actor types (sensors, actuators, inputs, outputs, tasks, and arbiters, respectively) and the state-memory actors.

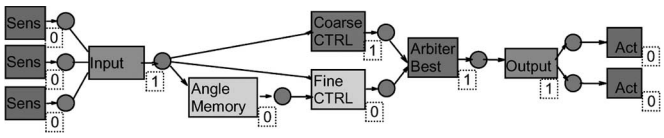


Fig. 6. Fine controller reads current and previous angle.

Sensor and actuator actors correspond to software drivers that read and update, respectively, the sensor and actuator devices interacting with the plant. Input actors perform sensor fusion, i.e., they read results coming from multiple sensors and compute a robust measurement of the quantity of interest by using some deterministic algorithm of designers' choice, e.g., the median value. With sensor fusion, designers can sample the quantity of interest by using sensors with different precision, accuracy, cost, and reliability. Output actors are used to balance the load on the actuators. Task actors perform general computation. Arbiter actors are similar in function to input actors: They fuse values that come from actors with different criticality and that must reach the same output actor[5] [e.g., braking command and antiblocking system (ABS)]. Sometimes, arbiters implement fixed-priority multiplexers like in the pendulum example. Finally, state memories are special actors that operate as one-reaction delays, thus expressing data dependences between events belonging to consecutive reactions.

Definition 2: Given an FTDF graph $\mathcal{G}$ and a vertex $v \in V$, $\mathrm{neig}^+(v) = \{w \in V \text{ s.t. } (v, w) \in E\}$ is the set of successor neighbors of v, and $\mathrm{neig}^-(v) = \{w \in V \text{ s.t. } (w, v) \in E\}$ is the set of predecessor neighbors of v. The set $\mathrm{neig}(v)$ of all the neighbors of v is the union of the two sets.

Clearly, $v \in M \Rightarrow \mathrm{neig}(v) \subset A$, and symmetrically, $v \notin M \Rightarrow \mathrm{neig}(v) \subseteq M$ because $\mathcal{G}$ is bipartite. The definition extends to sets in the natural way: $\mathrm{neig}(S) = \cup_{v \in S} \mathrm{neig}(v)$.

Definition 3: Given an FTDF graph $\mathcal{G}$ and an actor $v \in A$, the successor actors of v are $\mathrm{succ}(v) = \mathrm{neig}^+(\mathrm{neig}^+(v))$, and the predecessor actors are $\mathrm{pred}(v) = \mathrm{neig}^-(\mathrm{neig}^-(v))$.

The following rules specify the set of valid actor compositions to obtain a legal FTDF graph.[6]

Definition 4: An FTDF graph $\mathcal{G}$ is "legal" if the following conditions are met.

1) $\mathcal{G}$ contains no causality cycles, i.e., if graph $\mathcal{G}' = (V', E')$, where $V' = V \setminus A_M$, and $E' = E \cap (V' \times V')$ is acyclic.
2) $\forall v \in A_I, \quad \mathrm{pred}(v) \subset A_S \cup A_M \wedge \forall v \in A_S, \quad$ and $\mathrm{succ}(v) \subset A_I$.
3) $\forall v \in A_{\mathrm{Act}}, \mathrm{pred}(v) \subset A_O \wedge \forall v \in A_O, \;$ and $\mathrm{succ}(v) \subset A_{\mathrm{Act}} \cup A_M$.
4) $\forall v \in A_S, \; \mathrm{neig}^-(v) = \emptyset$, and $\forall v \in A_{\mathrm{Act}}, \; \mathrm{neig}^+(v) = \emptyset$.

Ignoring the technicalities related to the use of memory actors, this definition states that a legal FTDF must start with sensor actors (A_S) and end with actuator actors (A_{Act}), the results of sensor actors must be combined using input actors

[5]Note that we advocate running nonsafety-critical tasks, e.g., door controllers, on separate hardware. However, some performance-enhancement tasks, e.g., lane-keeping and stability enhancements, may share sensors and actuators with critical tasks (steer-by-wire). It may be profitable to have them share the execution platform as well.

[6]Some basic rules (e.g., all input and output ports of an actor should be connected, data types should be matched, etc.) are common to most data-flow models and are assumed implicitly here.

(A_I), and actuator actors must be driven by output actors (A_O). Figs. 5 and 6 are examples of legal graphs.

Actors communicate by exchanging tokens over unidirectional (possibly many-to-many) communication media. Each token consists of two fields: 1) Data—the actual data it transports and 2) Valid—a Boolean flag indicating the outcome of fault detection on this token. When Valid is "false," either no data are available for this reaction or the available data are not correct. In both cases, the Data field is ignored. New tokens are produced whenever an active element (a regular actor or a memory actor) fires. Each actor has a Boolean firing rule which determines whether enough input tokens are valid for it to fire. Actors in $A_S \cup A_{\mathrm{Act}} \cup A_O \cup A_T \cup A_M$ always use the AND firing rule, i.e., they require all inputs. Designers may specify partial firing rules for input and arbiter actors. For example, in Fig. 5, the input actor reading data from the three sensors may produce a valid result even when one of the sensors cannot deliver data (e.g., if the sensor is mapped on a faulty ECU). Similarly, the arbiter actor may fire with just the bang-bang result available.

Actors fire in an order that satisfies the data dependences that are captured by the arcs in the FTDF graph $\mathcal{G}$. The synchronous semantics [8], [31], [32] implies that the firing activity occurs as a sequence of reactions. Before an actor (notably a sensor actor) can fire a second time, all the actors (including the actuator actors) must complete the current reaction. This behavior requires at most a single-place buffer for communication between actors.

Actors may be replicated for redundancy purposes: All replicas of the same source actor write to the same medium, and all destination actors read from it. Media act both as mergers and repeaters that deliver either the single "merged" result to all destinations or an invalid token if no correct result is determined during this reaction. This abstraction nicely encapsulates any fault-detection and fault-recovery mechanisms in the underlying execution platform. Using communication media, actors always receive exactly one token per input, possibly invalid, and the application behavior is independent of the type of faults in the execution platform. Based on the firing rule of each destination, they may or may not be able to fire when some inputs are invalid. An actor that fires executes its sequential code, which is as follows:

1) stateless (state must be stored in memory actors);
2) deterministic (identical inputs generate identical outputs);
3) nonblocking (once fired, it does not await for further tokens, data, or signals from other actors);
4) terminating (bounded WCET).

A memory provides its state at the beginning of a reaction and has a source actor, possibly replicated, that updates its state at every reaction. State memories are analogous to latches in a sequential digital circuit: They store the results produced during the current reaction for use in the next one. Actors that need to keep state across reactions must have one output and one input connected to a memory like in Fig. 5, where the fine controller computes the derivative of the pendulum position. Fig. 6 shows an alternative way to compute the derivative, where the fine controller reads both the current and the previous values of the pendulum position. Making memories external to regular actors simplifies the task of keeping them coherent after actor (and memory) replication and in the presence of faults.

How to implement a communication medium depends on the type of faults that can affect the arrival of input tokens at each period.[7] Communication media must be distributed to withstand platform faults. Typically, this distribution means having a repeater on each source ECU and a merger on each destination ECU. The number of potential messages between the redundant repeaters and mergers may become very large, but channel broadcasting helps reduce the traffic greatly.

B. Expressing Criticality and Redundancy

FTDFs are designed to be independent from the fault model, i.e., the type of faults that the execution platform may exhibit. Designers specify how the system should behave in the presence of such faults, labeling FTDF graphs with actor criticality $\chi : A \to \mathbb{N}$, as shown in Figs. 5 and 6. Criticality inversion happens when a critical actor needs input data from less critical actors whose execution may not be guaranteed for some failure patterns. A criticality assignment is strictly proper if, $\forall v \in A,\ \forall w \in \mathrm{pred}(v)\ \chi(w) \geq \chi(v)$.

Definition 5: A criticality assignment of a legal FTDF $\mathcal{G}$ is proper if, $\forall v \in A \setminus (A_I \cup A_A), \forall w \in \mathrm{pred}(v)\ \chi(w) \geq \chi(v)$.

Note that input and arbiter actors may have partial firing rules that allow execution when some or even all source actors are not providing data. This is the rationale for desiring at least a proper criticality assignment in practical cases.

FTDF graphs can express redundancy through the replication of one or more actors. All the replicas of an actor $v \in A$ are denoted by $\mathcal{R}(v) \subset A$. Note that any two actors in $\mathcal{R}(v)$ are of the same type and must compute the same function.[8]

Definition 6: An FTDF graph $\mathcal{G}$ is "redundant" when some actors are replicated, i.e., $\exists v \in A \text{ s.t. } \mathcal{R}(v) \neq \{v\}$.

IV. Fault-Tolerant Software Deployment

In this section, we first formalize the problem of correctly deploying an FTDF application on a distributed execution platform that can exhibit (both transient and permanent) faults. Then, we propose an algorithm to solve it automatically. Finally, we discuss some optimizations to improve the solution's quality. Our approach is based on introducing redundancy via software replication. We replicate both actors and data transmissions, and we map and schedule the replicas on resources that do not fail simultaneously. Hence, at least one replica of every critical computation/communication completes successfully and contributes to the "survival" of the feedback-control application. Software replication has its costs: program memory, CPU, and bus cycles. We rely on guidelines from the designer and replicate only computation and communication for the most critical subsets of the control application. Another option to reduce the run-time resource demands (CPU and bus cycles) is passive replication, which is discussed in [14].

[7] Assuming fail silence, merging amounts to selecting any of the valid results, typically the first one received; assuming value errors, majority voting is necessary on the nominally identical results of the different replicas of the source actor; assuming Byzantine faults, we need rounds of voting (consensus problem [33]). If a majority cannot be elected, then the medium presents an invalid token to all destinations.

[8] This rule is motivated in Section V-A where we explain replica determinism.

A. System Specification

The various inputs for the SCRAPE design flow (Fig. 2) can be captured formally in a system-specification tuple $(\mathcal{G}, T_{\max}, \mathrm{PG}, F, \chi, \psi, \mu, \tau)$, where the variables are defined as follows.

1) $\mathcal{G}$ is the FTDF graph, as of Definition 1, specifying the control algorithms and $T_{\max}$ is its execution period.
2) $\mathrm{PG} = (P, C, D)$ is the platform graph that specifies the topology and connectivity of the execution platform, where P is the set of ECUs, C is the set of channels, and $D \subset P \times C$ is the set of edges connecting them.
3) $F \subset 2^{P \cup C}$ is the finite set of failure patterns including the empty failure pattern, e.g., set $F = \{\emptyset, f_1, \ldots, f_K\}$ contains $K + 1$ failure patterns. The faults to be tolerated can be derived through statistical analysis.
4) The fault behavior (χ, ψ) specifies which tasks should be guaranteed execution under the different failure patterns in F, where $\chi\colon A \to \mathbb{N}$, $\psi\colon F \to \mathbb{N}$ label actors and architecture components with a criticality. Given a set of faults $f_o \subset P \cup C$, let $F_o = \{f \in F, \text{s.t. } f_o \subseteq f\}$ be the covering failure patterns and $\psi_o = \min \psi(F_o)$ the minimum criticality. Then, the fault behavior requires that at least one replica of each actor a, such that $\chi(a) \geq \psi_o$, be executed when the set of faults f_o occurs.
5) $\mu\colon V \to 2^{P \cup C}$ is the mapping-constraint function that specifies on which vertices of PG can a given vertex of $\mathcal{G}$ be mapped. Some actors may require special resources that are not available at all ECUs. For instance, the sensor and actuator actors need direct access to the I/O resources. Also, μ can be used to guide the synthesis tool. In Section VI, we illustrate how designers can use the specification tuple to improve the synthesis results interactively.
6) $\tau\colon V \times P \cup C \to \mathbb{N}$ is the performance annotation.

These are used to specify the estimated WCET and worst-case transmission time (WCTT) of actors on ECUs and communication on channels.

Running Example. For brevity, we do not offer the formal description of $\mathcal{G}$ and PG for the inverted-pendulum controller, but rather refer to Figs. 5 and 2, respectively. Let us consider the failure of at most one ECU at a time: $F = \{\emptyset, \{\text{ECU0}\}, \{\text{ECU1}\}, \{\text{ECU2}\}\}$. An upper bound on the probability of system failure is given by the probability that any of the fault combinations not in F occurs: $\bar{F} = 2^{P \cup C} \setminus F = 2^{\{\text{ECU0, ECU1, ECU2, CH0, CH1}\}} \setminus F$. If this value is too high, we may need to move some of the elements of $\bar{F}$ into F or perform a more detailed analysis, e.g., based on fault trees [34]. Correspondingly, the desired fault behavior (χ, ψ) is

$$\chi(a) = \begin{cases} 0, & \text{if } a \in A_S \cup A_{\mathrm{Act}} \cup \{\text{fine CTRL}\} \\ 1, & \text{otherwise} \end{cases}$$

$$\psi(f) = \begin{cases} 0, & \text{if } f = \emptyset \\ 1, & \text{otherwise} \end{cases}.$$

Each ECU in Fig. 2 has access to one position sensor. ECU0 and ECU2 each have access to one of the two torque actuators.

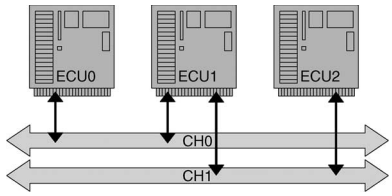


Fig. 7. Communication between ECU0 and ECU2 is routed via ECU1.

The mapping constraints indicate that the sensor and actuator processes execute only on their own ECU, i.e.,

v	Sens_i	Act0	Act1	$\in M$	otherwise
$\mu(v)$	$\{\text{ECU}_i\}$	ECU0	ECU2	$P \cup C$	P

For each process, function τ provides a finite WCET value for each ECU, where it can be executed, and the ∞ value otherwise.

B. Redundant Mapping and Scheduling

Given a set of control algorithms specified as an FTDF graph $\mathcal{G}$ and an execution platform graph PG, we use another graph to model the redundant allocation (mapping) of actors and communication on the execution platform.

A mapping of $\mathcal{G}$ on PG is a directed graph $\mathcal{L} = (L_V, L_E)$ whose vertices in L_V are elements of $(P \cup C) \times V$, where $V = A \cup M$ is the set of vertices of $\mathcal{G}$ and $P \cup C$ is the set of vertices of PG. The presence of a vertex $l \in L_V$, with $l = (r, v)$, means that actor or medium v is mapped to resource r. In a redundant mapping, the replicas of a same actor can be mapped to multiple resources. For a given actor or medium $v \in V$, the set $\ell(v) = \{r \in P \cup C, \text{ s.t. } (r, v) \in L_V\}$ denotes the set of vertices of PG where v is mapped. An edge $e \in L_E$, with $e = (l_1, l_2)$, where $l_1 = (r_1, v_1)$ and $l_2 = (r_2, v_2)$, models data transfer from l_1 to l_2. These data transfers may reflect one of the following two possible cases.

1) Two actors are mapped on a same ECU, and the first delivers data to the second; no channel is involved, i.e., $r_1 = r_2 \in P$ and $v_1, v_2 \in A$.
2) One actor mapped on an ECU transmits or receives data on a channel, i.e., $r_1 \in P, r_2 \in C, v_1 \in A, v_2 \in M$ or $r_1 \in C, r_2 \in P, v_1 \in M, v_2 \in P$.

A mapping must satisfy edge consistency: Edge $e \in L_E$ connects vertices $l_1 = (r_1, v_1)$ and $l_2 = (r_2, v_2)$ only if the associated FTDF elements depend on one another, i.e., if $(v_1 \in A \wedge v_2 \in \mathrm{succ}(v_1)) \wedge (r_1 = r_2 \in P)$ or if $((v_1, v_2) \in E) \wedge ((r_1, r_2) \in D \vee (r_2, r_1) \in D)$. These reflect the previous two cases of actor-to-actor communication on a same ECU through memory and the dependency between an actor and a communication medium.

Some platform graphs PG may need the use of routing to transmit data between pairs of ECUs. For instance, in Fig. 7, if two communicating actors are mapped on ECU0 and ECU2, respectively, then we need an additional routing actor on ECU1 to relay the message from channel CH0 to channel CH1. The redundant mapping may contain such routing actors. Routing actors may introduce causality cycles, as discussed in [35]. However, these cycles are only apparent: As soon as one of the source tasks generates the data, this enables at least one of the routing actors in the cycle, thus enabling all the others. In fact,

each routing actor has only one input communication medium, and it can fire as soon as at least one of its source actors produces the data. Platforms with a buslike communication infrastructure do not need routing actors because all ECUs are connected to the same set of buses. Because the redundant mapping $\mathcal{L}$ preserves the dependences in the application $\mathcal{G}$, it contains no causality cycles. Therefore, if we neglect memory actors, $\mathcal{L}$ defines a partial order.

A schedule $\mathcal{S}$ is defined as a pair of functions $(g(\cdot), h(\cdot))$, with $g : P \to \bar{A}^*$ and $h : C \to \bar{M}^*$, where $\bar{A}^*$ and $\bar{M}^*$ are the sets of ordered subsets of A and M, respectively. For each ECU $p \in P$, $g(p)$ denotes the sequence of actors that run on p, thereby defining a total order on actors mapped on p. Similarly, $\forall c \in C,\ h(c)$ defines a total order on data communication mapped on channel c. A pair $(\mathcal{L}, \mathcal{S})$ is called a deployment.

To avoid deadlock, the total orders defined by $\mathcal{S}$ must be compatible with the partial order in $\mathcal{L}$. To avoid causality problems, memory actors execute before any other actor, thus using the results of the previous reaction. Schedules based on total orders are called static: There are no run-time decisions to make, each ECU and each channel controller simply follows the schedule. However, in the context of a faulty execution platform, an actor may not receive enough valid inputs to fire, and this lack of inputs may lead to starvation. Like in [23], we solve this problem by skipping an actor if it cannot fire and by skipping a communication if no data are available. We support both flexible static schedules and time-triggered static schedules: In the first model, as soon as we skip actors and communications, we evaluate the firing rule of the next actor or communication in the schedule; in the second one, we wait for the predetermined start time of the next actor or communication.

Given the system specification, a synthesis algorithm derives a fault-tolerant deployment, i.e., a redundant mapping $\mathcal{L}_{\mathrm{FT}}$ and its associated schedule $\mathcal{S}_{\mathrm{FT}}$.

Currently, the synthesis and verification algorithms are derived assuming fail silence, i.e., components provide either correct results or no results at all. This is a desirable fault behavior as it offers strong fault containment, i.e., faults do not propagate outside the faulty subsystem. Fail-silent very-large-scale-integration platforms can now be realized with limited area overhead and virtually no performance loss [36]. Software methods can provide good coverage too [37].

C. Mapping Synthesis

Before tackling the problem of synthesizing a redundant mapping $\mathcal{L}_{\mathrm{FT}}$, we consider an auxiliary mapping problem.

Problem 1: Given $\mathcal{G}$, PG, and a set of constraints μ', find an edge-consistent mapping $\mathcal{L}' = (L'_V, L'_E)$, such that the following conditions are satisfied.

1) $\forall v \in A, (\ell(v) = \{p\}) \wedge p \in \mu'(v)$.
2) $\forall v \in M, \ell(v) \subset \mu'(v)$.

A solution to Problem 1 is a nonfault-tolerant mapping. In fact, having $\ell(v) = \{p\}$ means that actor v is mapped to only one processor. We may map communication media to multiple channels to reach all destination actors, but Problem 1 does not require redundancy to tolerate channel faults. We integrated SynDex [41] into SCRAPE, and we use it to solve an instance of Problem 1 making sure that we do not map critical actors and communication on faulty components. This strategy effectively maps a same actor on multiple ECUs and hence achieves redundancy. Algorithm 1 uses the solutions of a number of instances of Problem 1 to synthesize a fault-tolerant mapping $\mathcal{L}_{\mathrm{FT}}$.

We start with the empty failure pattern $\emptyset$ and map the entire FTDF graph $\mathcal{G}$ on the whole architecture. Then, we consider each failure pattern $f \in F$ and solve a new instance of Problem 1. In Step 2a), we "mark" the faulty components to prevent mapping critical actors or communication on them. Because the execution of actors v with criticality $\chi(v) < \psi(f)$ does not need to be guaranteed, we constrain them on the same resources where they were mapped in Step 1). For critical actors/communication, we consider the mapping constraints $\mu(v)$ from the specification and remove the faulty components in f. After solving all nonfault-tolerant mappings, we build a redundant mapping. In Step 3), we obtain the redundant mapping as follows:

$$L'_{V_{\mathrm{FT}}} = \bigcup_{f \in F} L_{V_f} \quad L'_{E_{\mathrm{FT}}} = \bigcup_{f \in F} L_{E_f}$$

where the union runs on all failure patterns in F, including the empty failure pattern $\emptyset$. At the end, for each failure pattern, we have at least one copy of every critical actor on some nonfaulty resource. Due to the greedy nature of the algorithm, the amount of replication depends on the order used to examine the failure patterns in F and is not guaranteed to be minimum.

Algorithm 1: Consider tuple $(\mathcal{G}, T_{\max}, \mathrm{PG}, F, \chi, \psi, \mu, \tau)$
1) let $\mathcal{L}_\emptyset$ be the solution to Problem 1 using $\mu' \equiv \mu$
2) for each $f \in F \setminus \{\emptyset\}$ do
a) build a set of constraints μ_f such that

$$\mu_f(v) = \begin{cases} \ell_\emptyset(v), & \text{if } (v \in A) \wedge \chi(v) < \psi(f) \\ \mu(v) \setminus f, & \text{otherwise} \end{cases}$$

b) let $\mathcal{L}_f$ be the solution to Problem 1 using $\mu' \equiv \mu_f$
3) merge the resulting mappings into a redundant mapping
$\mathcal{L}'_{FT} = \bigcup_{f \in F} \quad \mathcal{L}_f = (L_{V_{\mathrm{FT}}}, L_{E_{\mathrm{FT}}})$

D. Redundant Mapping Transformation

Simply merging all the mappings relative to the failure patterns gives a raw redundant mapping graph. This graph can be transformed with some basic heuristics such as: 1) adding/pruning dependency edges in L_E or 2) adding/pruning entire redundant communication paths, including vertices in L_V. For instance, assume that actor v is mapped on ECU r (therefore, there is a vertex $(r, v) \in L_V$) and that the same input data are available both from memory (data sent by some other actor v' on the same ECU r) and from some dependency node (data sent by actor v' running on other ECUs). We could directly get data from memory instead of from the dependency node. Pruning the edge from the dependency node could potentially save data transmission time on the bus. Furthermore, if the dependency node only had one destination ECU and we removed this edge, then we can remove the dependency node too. While this choice may produce lower bus bandwidth, pruning may, in general, reduce redundancy below the minimum required to meet the fault behavior. The example of previous pruning may seem safe because if ECU r is

available to fire v, it was also available to fire v'; hence, its result will be in memory. This assumption is often true, but actor v' may not be able to fire on ECU r because it could be waiting for some inputs from other actors scheduled on some faulty ECU. In this case, having the result of v' available also from the bus may provide better fault tolerance. Moreover, depending on the execution model, the designer might want to preserve both data paths. For example, if we allow a dynamic run-time execution model, we can abort execution of actor v' on r when we receive the same result from the bus. Then, it might be advantageous to have both paths because we could start the execution of v on r as soon as the result of v' is known, either from the bus or from completing execution of v' on r.

By adding new communication paths for the data dependences, we can provide reliable data delivery in spite of faults. We use this mechanism to tackle replica determinism (see Section V-A). By adding edges between nodes on a same resource, we can introduce precedence constraints, thus limiting the number of possible total orders and guiding the selection of a schedule $\mathcal{S}_{\mathrm{FT}}$. In general, adding edges and communication paths does not compromise fault tolerance because it means additional redundancy. If, instead, we choose to prune some edges and nodes, we must run a verification tool to check whether the solution still meets the fault behavior.

E. Scheduling the Redundant Mapping

Using the mapping graph and the set of firing rules, we find all predecessor firings required for the firing of each pair of actor/ECU or data dependency/channel. However, the mapping graph itself is not the final schedule yet because it only gives a partial order. To obtain a fault-tolerant deployment $(\mathcal{L}_{\mathrm{FT}}, \mathcal{S}_{\mathrm{FT}})$, we derive a schedule $\mathcal{S}_{\mathrm{FT}}$ for the execution of $\mathcal{L}_{\mathrm{FT}}$. More formally, for each ECU $p \in P$ and for each channel $c \in C$, we derive a total order that is compatible with $\mathcal{L}_{\mathrm{FT}}$ for the execution of actors mapped on p, i.e., of actors in $\{v \in V,\ \text{s.t. } p \in L(v)\}$ and for the transmission of data mapped on $c \in L(m)$.

If we did not prune the redundant mapping and if there is sufficient redundancy in the execution platform, the resulting fault-tolerant deployment $(\mathcal{L}_{\mathrm{FT}}, \mathcal{S}_{\mathrm{FT}})$ is guaranteed by construction to meet the fault behavior $(\chi(\cdot), \psi(\cdot))$. We can derive the schedule $\mathcal{S}_{\mathrm{FT}}$ using a list-scheduling algorithm driven by any heuristic cost function [38]. Heuristics minimizing the worst-case reaction time are excellent candidates.

Running Example. For brevity, we do not describe the resulting fault-tolerant deployment in terms of the graph $\mathcal{L}_{\mathrm{FT}}$ and its associated schedule $\mathcal{S}_{\mathrm{FT}}$. Rather, we refer to Fig. 3(a) and its description in Section II-E. Notice that the arbiter process ("C0AR1b") running on ECU2 waits for the result of the bang-bang controller ("C0FUNc") on channel1. To improve the schedule, we can force the scheduler to prioritize data communication using additional precedence constraints

$$\#\,\texttt{Bus channel1}: \quad \texttt{C0FUNc!o0} \quad \texttt{C1FUNf!o0}.$$

This specifies that data communication "C0FUNc!o0," i.e., the output "o0" of actor "C0FUNc," must precede "C1FUNf!o0" on channel1. Since this precedence constraint does not create a cycle in the redundant mapping, it can be safely used in the topological sort algorithm. Fig. 3(b) shows the corresponding synthesized solution with a reaction time reduction from 355 to 260 time units.

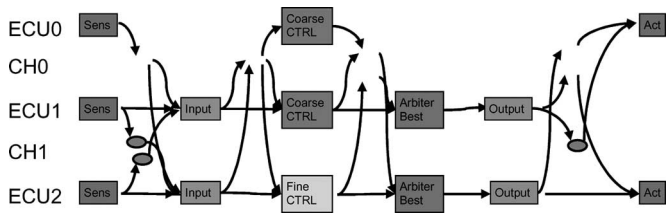


Fig. 8. Failure of channel CH0 causes the two replicas of the Arbiter actor to receive different subsets of the inputs.

V. Verification

We develop a tool to validate the functionality of the control application by simulating the unmapped FTDF graph $\mathcal{G}$. The simulation supports replicated actors and the injection of omission errors. Furthermore, for each deployment, we can statically check two important properties related to both functionality and timing: replica determinism and worst-case reaction time. Because these checks are done for each failure pattern, we incidentally also verify adherence to the fault behavior.

A. Replica Determinism

Given a mapping $\mathcal{L}$, we must preserve replica determinism: If two actors in $\mathcal{R}(v)$ (i.e., replicas of a same actor v) fire, they produce identical results. For general MoCs, the order of arrival of results must also be the same for all replicas. Synchrony of FTDF makes this check unnecessary. Replica determinism in FTDF can be achieved with the enforcement of the following two conditions.

1) All actors in $\mathcal{R}(v)$ compute the same function.
2) For any failure pattern, if two replicas get a firing subset of inputs, they get the same subset of inputs.

Condition 1) is enforced by construction as only identical replicas are allowed. Condition 2) amounts to a consensus problem, and it can be either checked at run time (like for Byzantine agreement rounds of voting) or analyzed statically at compile time (if the fault model is milder). Our interest in detectably faulty execution platforms makes the latter approach appear more promising and economical. Condition 2) is trivially true for all actors with the "AND firing rule." For input and arbiter actors, the condition must be checked and enforced.

Running Example. In Fig. 8, both replicas of the Input actor receive the same subset of data from the two sensors Sens on ECU1 and ECU2. Normally, they fire, producing the same results from the same data. However, if channel CH0 fails, the two replicas of the Arbiter actor get different subsets of the inputs: The replica on ECU1 only gets the result from the Coarse CTRL (controller) actor, and the replica on ECU2 only gets the result from the Fine CTRL actor. Hence, if they fire, they will produce different results.

We derive procedure extend$(\mathcal{L})$ that transforms a mapping $\mathcal{L}$ to enforce Condition 2). Its basic step is the following.

- If Condition 2 fails, some data are produced but delivered only to some replicas. Then, extend the mapping with routings of the results to the replica that lacks them.

If there is enough connectivity in PG, repeating this step will stabilize the mapping and achieve replica determinism.

B. Timing Analysis

In the following, a "task" indicates a vertex of the redundant mapping graph $\mathcal{L}_{\rm FT}$. Given a deployment $(\mathcal{L}_{\rm FT}, \mathcal{S}_{\rm FT})$, we compute the time-out for each task, and the worst-case reaction time from sensors to actuators, for all failure patterns. We illustrate the algorithm assuming fail silence, but it can be extended for the case of majority voting or rounds of voting. While we illustrate the case of a flexible static execution model (see Section IV-B), the time-triggered version is simply derived taking maxima across failure patterns. Before a task can fire, it needs to wait for the outputs generated by its predecessor tasks and for the resource to be ready.

Consider a vertex $l = (r, v)$ in $\mathcal{L}_{\rm FT}$. Assume that v has N input ports ($N = 1$ if $v \in M$). We compute, for each failure pattern $f \in F$, the following values for task l:

1) the availability time of each input $t_{I_j}(l, f)$;
2) the time at which it is enabled $t_n(l, f)$;
3) the time to fire (start) $t_s(l, f)$;
4) time-out $t_o(l)$;
5) the time to results (completion) $t_c(l, f)$;
6) the time to end $t_e(l, f)$.

For the jth input port, task l can receive replicated data from M_j tasks. Let l_z, with $z = 1, \ldots, M_j$, denote these source tasks in $\mathcal{L}_{\rm FT}$. Let $t_c(l_z, f)$ be the time that the l_z task produces data in failure pattern f. For some f, $t_c(l_z, f)$ can be ∞.

Given fail silence, we define input $t_{I_j}(l, f)$ to be the (earliest) time the jth input port receives data in failure pattern f

$$t_{I_j}(l, f) = \min_{z=1}^{M_j} t_c(l_z, f). \tag{1}$$

If no result is available for some f, then $t_{I_j}(l, f) = \infty$.

Let $t_n(l, f)$ be the time when the task is enabled. Tasks with the "AND" firing rule need all inputs

$$t_n(l, f) = \max_{j=1}^{N} t_{I_j}(l, f). \tag{2}$$

For "partial" firing rules, we define the enabling function $\mathrm{en}()$

$$\mathrm{en}(l, t_{I1}, \ldots, t_{IN}) \begin{cases} \overline{\max}_{j=1}^{N}\, t_{I_j}, & \text{if firing subset} \\ \infty, & \text{otherwise} \end{cases}$$

where t_{I_j} is the arrival time of input I_j, and $\overline{\max}$ takes the finite maximum in a finite set $E \subset (\mathbb{R}_0^+ \cup \{\infty\})$. If all values are ∞, then $\overline{\max}$ returns zero by default, i.e.,

$$\overline{\max}\, E = \max\left\{\{0\} \cup \{e \in E \text{ such that } e \neq \infty\}\right\}. \tag{3}$$

For example, for the "n-out-of-N" firing rule, we have

$$\mathrm{en}(l, t_{I1}, \ldots, t_{IN}) = \begin{cases} \overline{\max}_{j=1}^{N}\, t_{I_j}, & \text{if } |\{t_{I_j} \neq \infty,\ j = 1, \ldots, N\}| \geq n \\ \infty, & \text{otherwise.} \end{cases}$$

By looking at $\mathrm{en}(\cdot)$, we know if a task could fire in a given failure pattern. However, at run time, we do not know f, and before firing with missing inputs, we must wait for a time-out

$$t_w(l) = \overline{\max}_{f \in F}\, \mathrm{en}\left(l, t_{I1}(l, f), \ldots, t_{IN}(l, f)\right).$$

Then, for failure pattern f, the time when l is ready to run, and when no more inputs are going to arrive, is denoted by

$$t_n(l, f) = \max\left(\mathrm{en}\left(l, t_{I1}(l, f), \ldots, t_{IN}(l, f)\right), t_w(l)\right). \tag{4}$$

If, for some f, we have $t_n(l, f) = \infty$, not enough input ports receive data, and the task is not enabled to fire.

We define the time-out to be "the latest time when the task can be ready to fire under any failure pattern"

$$t_o(l) = \overline{\max}_{f \in F}\, t_n(l, f). \tag{5}$$

If time exceeds $t_o(l)$, no future input will enable the task. In order to fire v, resource r should be available. Let $l' = (r, v') \in \mathcal{L}_{\rm FT}$ such that v' immediately precedes v on resource r in the schedule $\mathcal{S}_{\rm FT}$. Let $t_e(l', f)$ be the time when task l' releases resource r. Then, task l starts execution at time

$$t_s(l, f) = \max\left(t_n(l, f), t_e(l', f)\right). \tag{6}$$

Task l will not fire under failure pattern f if $t_s(l, f) = \infty$. The results from this task are available in failure pattern f at time $t_c(l, f) = t_s(l, f) + \tau^*(l, f)$, where $\tau^*(l, f)$ is the execution time of v on resource r in failure pattern f. Notice that $\tau^*(l, f)$ can take one of two possible values. If the resource is not faulty, i.e., $r \notin f$, then $\tau^*(l, f) = \tau(v, r)$; otherwise, $\tau^*(l, f) = \infty$. Finally, the resource release time $t_e(l, f)$ is

$$t_e(l, f) = \begin{cases} \infty, & \text{if } r \in f \\ t_c(l, f), & \text{if } t_c(l, f) \neq \infty \\ \max\left(t_o(l), t_e(l', f)\right), & \text{otherwise.} \end{cases} \tag{7}$$

If the resource is faulty, it is never available. If the task generates results (hence $r \notin f$), the resource is released at $t_c(l, f)$. Otherwise, r does not fail, and l does not fire (due to lack of inputs). Hence, r can be released after the time-out $t_o(l)$ and after the release time $t_e(l', f)$ of the previous task.

We perform this analysis, proceeding from sensors to actuators. For each task, we analyze all failure patterns before going to the next task. The complexity is clearly linear in the number of nodes in $\mathcal{L}_{\rm FT}$ and in the number of failure patterns.

The latest time to generate outputs for all final actuation tasks, under all failure patterns, is the worst-case reaction time

$$T_e(\mathcal{L}_{\rm FT}, \mathcal{S}_{\rm FT}) = \overline{\max}_{f \in F,\ l \in \mathcal{L}_{\rm FT}} t_c(l, f)$$

i.e., the worst-case execution time of the whole deployment.

VI. Two Case Studies From Automotive Industry

We completed the following two industrial case studies with SCRAPE.

1) A simplified drive-by-wire system that is developed in collaboration with BMW. It consists of braking, steering, force feedback on the steering wheel, and a supervisory controller to enhance vehicle stability.
2) A steer-by-wire system that is developed in collaboration with General Motors. It consists of steering, force feedback on the steering wheel, and a supervisory controller for vehicle stability.

Both systems are not actual products but model key features in principle, they involve different design aspects: In the BMW

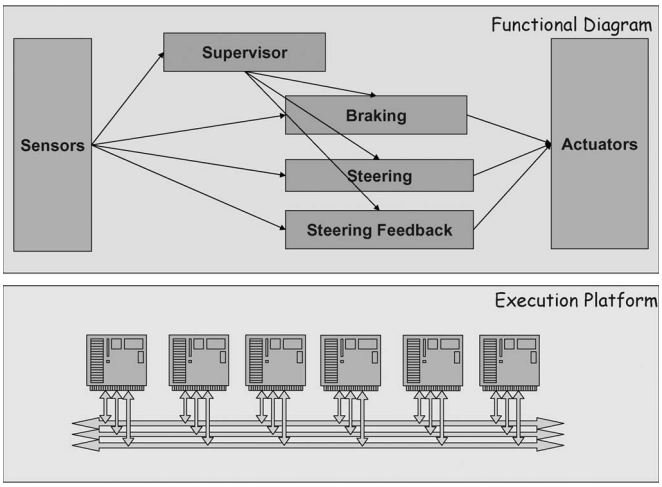


Fig. 9. Drive-by-wire system: Simplified functional diagram and platforms.

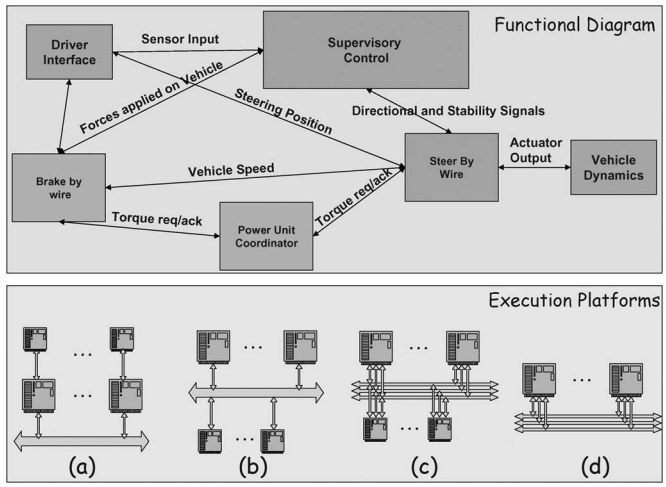


Fig. 11. Steer-by-wire system: Simplified functional diagram and platforms.

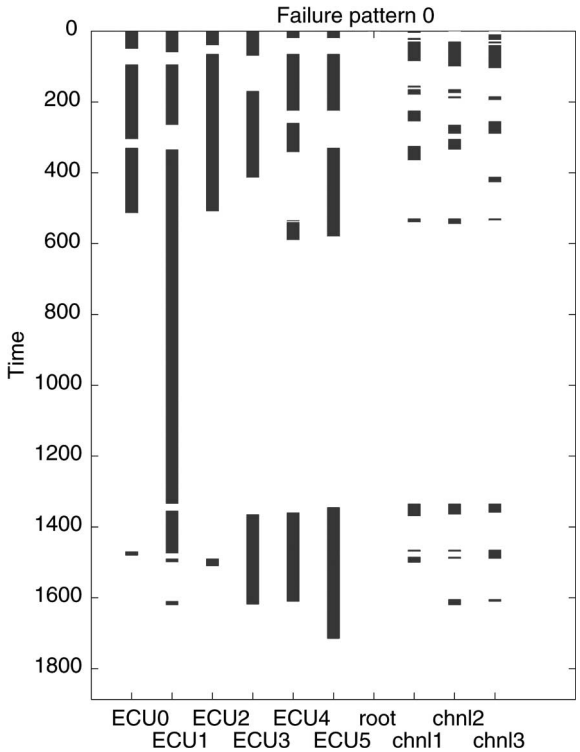


Fig. 10. Synthesized redundant mapping for the drive-by-wire example. To reduce clutter, the task names and data dependencies are removed.

	Description (see types in Figure 14)	**Mapping Constraints**
a1	type (a) (clustered asymmetric)	dedicated I/O ECUs
a2	type (a)	none
a3	type (b) (distributed asymmetric 1)	dedicated I/O ECUs
a4	type (b)	none
a5	type (b), fewer ECUs	dedicated I/O ECUs
a6	type (b), fewer ECUs	none
a7	type (c) (distributed asymmetric 2)	dedicated I/O ECUs
a8	type (c)	none
a9	type (d) (distributed symmetric), fewer ECUs	none
a10	type (d), fewer ECUs, dual bus failures possible	none
a11	type (c), fewer ECUs	none
a12	type (c), fewer ECUs, dual bus failures possible	none

Fig. 12. Alternative execution platforms for steer-by-wire system.

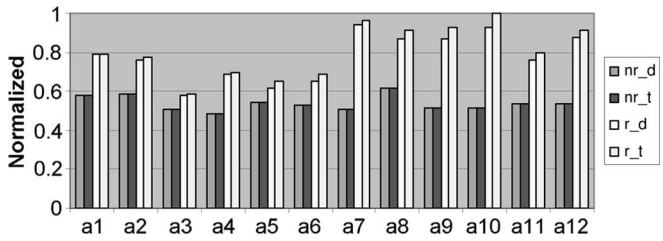


Fig. 13. Worst-case reaction time (normalized for IP protection).

case, we addressed a mix of critical requirements (e.g., the supervisor is less critical than other tasks), whereas in the General Motors case, we experimented a solution with a combination of fail silence and triple-modular redundancy (TMR).

A. *Drive-by-Wire Control System*

Fig. 9 (top) captures the basic functionality of the system. The following are the four main processes (tasks) involved.

1) *Braking* computes the four braking forces based on the brake pedal position and implements ABS control based on wheel speed.
2) *Steering* computes what force to apply to the front steering rack based on the steering wheel position.
3) *Steering feedback* computes the feedback torque to be applied to the steering wheel, based on the vehicle dynamics, later on acceleration, and on road conditions.
4) *Supervisory control* coordinates the other three processes in order to achieve and enhance vehicle stability based on data coming from accelerometers.

The first three processes are highly safety critical, i.e., their loss would lead to unacceptable consequences. Supervisory control adds value to the vehicle, but its loss is not safety critical because the driver may still act on the braking, steering, and throttle to stabilize the vehicle in case of a sudden change in the terrain conditions: The vehicle would still behave as a standard vehicle with ABS brakes. Hence, during the specification of the fault behavior (see Section II-D), the supervisor is marked as less critical, whereas integrity is required for the three critical processes. Fig. 9 shows the candidate execution platform. It

consists of six identical ECUs connected by three buses. Four ECUs are located next to the wheels, and two ECUs are found next to the driver commands. Together, they access many sensors/actuators while exploiting physical proximity to avoid a single failure point: This avoids the case where the loss of one ECU causes the loss of controllability of the system.

1) Synthesized Solution: The diagrams of Fig. 10 show the synthesis result.[9] They highlight a strong imbalance in the ECU workload, with ECU1 being the bottleneck: It is the most utilized among ECUs because it runs the supervisor actor. The worst-case reaction time is 1.7 ms, which is well within the desired control period of 5 ms. This is an acceptable result if the designers are satisfied with the implementation cost (and, e.g., with having spare room on the ECUs for future releases of the product or for reusing it across different vehicle models). In fact, because other ECUs are lightly loaded, designers may require the same level of integrity also for the supervisor (i.e., label the supervisor with the same criticality as the other three processes). This additional requirement should have a small impact on timing and virtually no impact on cost. Instead, if the production volume of this specific vehicle model is large enough to justify differentiating its execution platform from that of other vehicles, designers may seek a cheaper solution. Fig. 10 shows that reducing the performance of all ECUs but ECU1 would slightly increase the worst-case timing and could reduce costs. Similarly, if the solution had not met the timing constraints, then it would be useless to speed up the other ECUs without making ECU1 faster. Another option is breaking up the supervisor into smaller actors and trying to extract more parallelism. A finer granularity FTDF would, in general, yield a more balanced solution.

B. Steer-by-Wire Control System

Fig. 11 shows the simplified functional diagram of the steer-by-wire system. An interesting characteristic of this design is its interaction with the power unit coordinator. Because the electricity generated by the fuel cell powers both propulsion and the by-wire actuators, it is critical to coordinate its use in order to avoid dangerous fluctuations in the power grid. In our case, the power unit coordinator is assumed to be a predesigned module on a dedicated ECU.

We had a detailed FTDF graph specification for the steer-by-wire too: Aside from the interaction with the power unit coordinator, it models the steering and supervisor functionalities. The latter is specified in TMR by the designer, i.e., there are three explicit replicas of the supervisor, which will not be replicated further. A majority voter actor collects their results and elects a majority response. The voter actor, which appears in single copy in the specification, is automatically replicated during synthesis.

The main goal of this paper is to complete an exploration of the architectural space, i.e., the set of possible execution platforms that could support the application. We looked at a few basic alternatives including those shown in Fig. 11. In particular, the clustered architecture "a)" is made of a number of high-end ECUs connected to a high-speed and very reliable (low failure rate) bus. Some high-end ECUs also communicate to low-end ECUs using a slower and less reliable bus (with a failure rate that is of concern). The low-end ECUs process prevalently the system I/O (reading sensors and writing to actuators). In the distributed architectures [b), c), and d)], all ECUs communicate through a system of global buses. Architectures c) and d) have three parallel busses that are slower and less reliable than the one in b). For these architectures, we tried various configurations, including the following:

1) changing the number of high-speed ECUs;
2) preventing mapping of non-I/O tasks on low-end ECUs;
3) assuming very reliable components (i.e., only the empty failure pattern is specified in the fault behavior) versus considering single faults or even dual bus faults;
4) assuming dynamic versus static execution model.

For each configuration in Fig. 12, we run four syntheses (with/without redundancy and static/dynamic schedule). Given the large design space, we fully relied on automatic solution without providing any hints to our tool. The total run time for the 46 syntheses was less than 2 min on a Pentium mobile laptop running at 1.6 GHz. Figs. 13 and 14(a) and (b) show the following results, respectively:

1) safe period, i.e., worst-case duration of a reaction (from sensors to actuators) under all failure patterns;
2) average, minimum, and maximum CPU utilizations;
3) average, minimum, and maximum bus utilizations.

Note that the syntheses with no redundancy (denoted by "_nr" in the pictures) are there as a baseline comparison. The safe period does not increase dramatically when we introduce redundancy (denoted by "_r"), except for the distributed architectures with a slow bus system (from a7 to a12). Remarkably, the use of static (time-triggered) instead of dynamic execution models (static is denoted by "_t" and dynamic by "_d") does not affect the safe period, except when the system is heavily loaded. However, even for highly utilized systems, dynamic execution is not much faster than static. One characteristic that may help explain this limited advantage is that, in the steer-by-wire case, we did not mark any process as less critical, thereby executing all processes in the various failure patterns.

The results show that execution platforms with a slower bus system (starting with a7) have higher bus utilization. For solutions between a3 and a6, there is only a single fast bus, so the average, minimum, and maximum utilizations obviously coincide. Finally, the results corresponding to alternative a10 produce a remarkably well balanced (minimum close to maximum) utilization for both CPUs and buses.

C. Discussion

These experiments show how SCRAPE can explore quickly many alternative execution platforms along the following axes:

1) architecture: topology and number of components;
2) performance of the various components;
3) reliability of the various components (as reflected in the set of failure patterns to be considered);
4) mapping constraints, which may reflect the unavailability of binaries of some actors for some ECUs.

In these experiments, we did not explore the functionality space, e.g., introduction of pipelining and/or changes of the

[9]Because the diagram is good for assessing the overall quality of the solution, but it is far less practical to read out detailed information, the tool outputs the latter also in textual form.

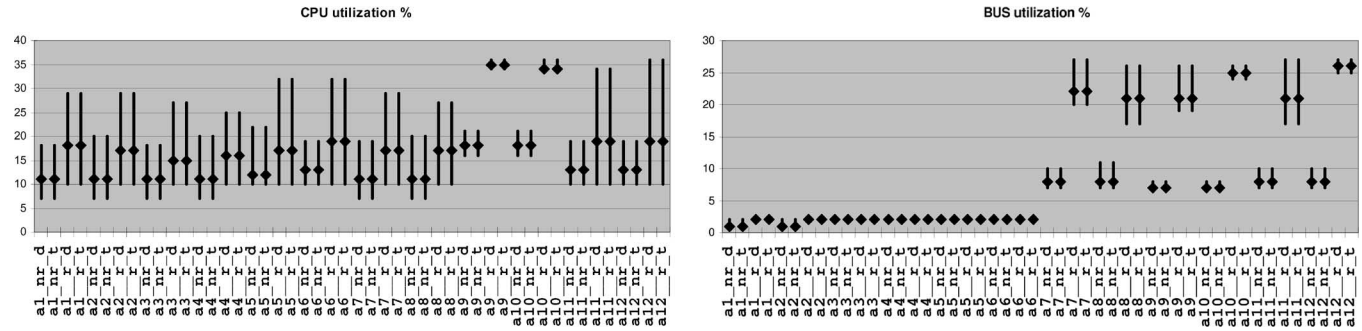


Fig. 14. Results for steer-by-wire control systems: CPU utilization and Bus utilization.

granularity of the FTDF graph to expose more parallelism. Besides comparing execution platforms based on the timing information, General Motors was interested in other metrics that are more specific to the automotive industry such as component reusability for larger economies of scale. Clearly, SCRAPE can be extended to support this kind of design exploration.

VII. CONCLUDING REMARKS

Designing cost-sensitive real-time control systems for safety-critical applications requires a careful analysis of the cost/coverage tradeoffs of fault-tolerant solutions. This further complicates the difficult task of deploying the embedded software that implements the control algorithms on the execution platform, which is often distributed around the plant (as it is typical, for instance, in automotive applications). Control theorists design the periodic control laws that run on the execution platform (composed of the distributed hardware and operating system, middleware, and drivers). These control laws address faults in the controlled plant (e.g., a flat tire or a stuck brake). However, in order to guarantee the end user safety, designers must deal also with faults in the execution platform.

We defined a new design methodology for safety-critical applications that advocates separation of concerns, and we use it to develop SCRAPE an interactive software environment where designers specify the functionality separately and independently from the execution platform and the faults that it may exhibit. The different parts of the specification, i.e., functionality, execution platform, and fault model, are processed together to automatically derive the fault-tolerant deployment of the embedded control software. This approach relieves designers from the burden of specifying and implementing detailed fault-tolerant mechanisms. Furthermore, it allows them to explore rapidly the design space, so that they can make informed decisions about changing the control algorithms, restructuring the execution platform, and refining its fault model.

SCRAPE has its foundation on FTDF and was realized assembling a new set of tools for fault-tolerant deployment. FTDF is a novel MoC that we propose for programming safety-critical control applications. FTDF deals with redundancy explicitly and is fault-model independent, i.e., it can be retargeted to execution platforms exhibiting fail silence, random errors, or more general error behavior. The fault-tolerant deployment tools, which include tools for redundant mapping and execution scheduling, currently support fail-silent execution platforms and platforms that produce detectably faulty results. We targeted real-time feedback-control applications, with no dynamic creation and dispatching of new tasks (i.e., the task workload is known statically at design time), and we support both permanent and transient platform faults. We used SCRAPE to explore the design space for two modern automotive applications. In particular, for a simplified steer-by-wire system under development at General Motors, we were able to compare 46 design alternatives in less than 2 minutes.

ACKNOWLEDGMENT

The authors would like to thank T. Demmeler of BMW Technology Office and S. Kanajan of General Motors for their collaboration in developing the drive-by-wire and the steer-by-wire systems, respectively, and C. Dima and A. Girault for inspiration and support.

REFERENCES

[1] E. A. Lee, "What's ahead for embedded software?" *Computer*, vol. 33, no. 9, pp. 18–26, Sep. 2000.

[2] D. Tennenhouse, "Proactive computing," *Commun. ACM*, vol. 43, no. 5, pp. 43–50, May 2000.

[3] H. Gill, "Challenges for critical embedded systems," in *Proc. 10th IEEE Int. Work. Object-Oriented Real-Time Dependable Syst.*, Sedona, AZ, Feb. 2005, pp. 7–9.

[4] A. Sangiovanni-Vincentelli, "Electronic-system design in the automobile industry," *IEEE Micro*, vol. 23, no. 3, pp. 8–18, May/Jun. 2003.

[5] E. A. Lee, "Absolutely positively on time: What would it take?"*Computer*, vol. 38, no. 7, pp. 85–87, Jul. 2005.

[6] R. Isermann, R. Schwarz, and S. Stolzl, "Fault-tolerant drive-by-wire systems," *IEEE Control Syst. Mag.*, vol. 22, no. 5, pp. 64–81, Oct. 2002.

[7] J. Laprie, Ed., *Depend Ability: Basic Concepts and Terminology in English, French, German, Italian and Japanese*, ser. Dependable Computing and Fault-Tolerant Systems, vol. 5. New York: Springer-Verlag, 1992.

[8] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The synchronous language twelve years later," *Proc. IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003.

[9] R. Alur *et al.*, "Hierarchical modeling and analysis of embedded systems," *Proc. IEEE*, vol. 91, no. 1, pp. 11–28, Jan. 2003.

[10] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal methods, validation and synthesis," *Proc. IEEE*, vol. 85, no. 3, pp. 266–290, Mar. 1997.

[11] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proc. IEEE*, vol. 91, no. 1, pp. 145–164, Jan. 2003.

[12] H. Kopetz and D. Millinger, "The transparent implementation of fault tolerance in the time-triggered architecture," in *Proc. Dependable Comput. Critical Appl.*, San Jose, CA, 1999, p. 191.

[13] C. Pinello, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, "Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications," in *Proc. Conf. Des., Autom. Test Eur.*, Feb. 2004, pp. 1164–1169.

[14] C. Pinello, "Design of safety-critical applications, a synthesis approach," Ph.D. dissertation, Univ. California, Berkeley, Aug. 2004.
[15] F. Cristian, "Understanding fault-tolerant distributed systems," *Commun. ACM*, vol. 34, no. 2, pp. 56–78, Feb. 1991.
[16] P. Jalote, *Fault Tolerance in Distributed Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1994.
[17] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Design optimization of time- and cost-constrained fault-tolerant distributed embedded systems," in *Proc. Conf. Des., Autom. Test Eur.*, 2005, pp. 864–869.
[18] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Synthesis of fault-tolerant schedules with transparency/performance trade-offs for distributed embedded systems," in *Proc. Conf. Des., Autom. Test Eur.*, 2006, pp. 706–711.
[19] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Synthesis of fault-tolerant embedded systems with checkpointing and replication," in *Proc. 3rd IEEE Int. Workhop Electron. Des., Test Appl.*, 2006, pp. 440–447.
[20] H. Kopetz and G. Grunsteidl, "TTP: A protocol for fault-tolerant real-time systems," *Computer*, vol. 27, no. 1, pp. 14–23, Jan. 1994.
[21] B. M. N. Kandasamy and J. P. Hayes, "Transparent recovery from intermittent faults in time-triggered distributed systems," *IEEE Trans. Comput.*, vol. 52, no. 2, pp. 113–125, Feb. 2003.
[22] T. Grandpierre, C. Lavarenne, and Y. Sorel, "Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors," in *Proc. 7th Int. Workshop Hardware/Software Co-Des.*, May 1999, pp. 74–78.
[23] C. Dima, A. Girault, C. Lavarenne, and Y. Sorel, "Off-line real-time fault-tolerant scheduling," in *Proc. Euromicro*, Mantova, Italy, Feb. 2001, pp. 410–417.
[24] A. J. Wellings, L. Beus-Dukic, and D. Powell, "Real-time scheduling in a generic fault-tolerant architecture," in *Proc. RTSS*, Dec. 1998, p. 390.
[25] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Embedded control systems development with GIOTTO," in *Proc. Languages, Compilers, Tools Embedded Syst.*, 2001, pp. 64–72.
[26] A. Casavola and E. Garone, "Adaptive fault tolerant actuator allocation for overactuated plants," in *Proc. 26th Am. Control Conf.*, New York, NY, Jul. 2007, pp. 3985–3990.
[27] H. Siu, Y. Chin, and W. Yang, "Reaching strong consensus in the presence of mixed failure types," *Inf. Sci.*, vol. 108, no. 1, pp. 157–180, Jul. 1998.
[28] J. Engblom and B. Jonsson, "Processor pipelines and their properties for static WCET analysis," in *Proc. 2nd. Int. Conf. EMSOFT*, 2002, vol. 2491, pp. 334–348.
[29] C. Ferdinand *et al.*, "Reliable and precise WCET determination for a real-life processor," in *Proc. 2nd. Int. Conf. EMSOFT*, 2001, vol. 2211, pp. 469–485.
[30] J. Dennis, "Data flow supercomputers," *Computer*, vol. 13, no. 11, pp. 48–56, Nov. 1980.
[31] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.
[32] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proc. IEEE*, vol. 79, no. 9, pp. 1270–1282, Sep. 1991.
[33] M. Barborak, M. Malek, and A. Dahbura, "The consensus problem in fault-tolerant computing," *ACM Comput. Surv.*, vol. 25, no. 2, pp. 171–220, Jun. 1993.
[34] M. L. McKelvin, G. Eirea, C. Pinello, S. Kanajan, and A. L. Sangiovanni-Vincentelli, "A formal approach to fault tree synthesis for the analysis of distributed fault tolerant systems," in *Proc. Conf. Embedded Softw.*, Sep. 2005, pp. 237–246.
[35] C. Dima, A. Girault, and Y. Sorel, "Static fault-tolerant real-time scheduling with "pseudo-topological" orders," in *Proc. FORMATS/FTRTFT*, 2004, pp. 215–230.
[36] M. Baleani, "Fault-tolerant platforms for automotive safety-critical applications," in *Proc. Int. Conf. Compilers, Archit. Synthesis Embedded Syst.*, 2003, pp. 170–177.
[37] F. Brasileiro, P. Ezhilchelvan, S. Shrivastava, N. Speirs, and S. Tao, "Implementing fail-silent nodes for distributed systems," *IEEE Trans. Comput.*, vol. 45, no. 11, pp. 1226–1238, Nov. 1996.
[38] T. Yang and A. Gerasoulis, "List scheduling with and without communication delays," *Parallel Comput.*, vol. 19, no. 12, pp. 1321–1344, Dec. 1993.
[39] K. Ahn, J. Kim, and S. Hong, "Fault-tolerant real-time scheduling using passive replicas," in *Proc. Pacific Rim Int. Symp. Fault-Tolerant Syst.*, 1997, pp. 98–103.
[40] S. Ghosh, R. Melhem, and D. Mosse, "Fault-tolerant scheduling on a hard real-time multiprocessor system," in *Proc. 8th Int. Parallel Process. Symp.*, Los Alamitos, CA, 1994, pp. 775–782.
[41] INRIA, *SynDEx Webpage*. [Online]. Available: http://www-rocq.inria.fr/syndex/

Claudio Pinello received the Laurea degree (*summa cum laude*) in electrical engineering from the Università di Roma, La Sapienza, Italy, in 1997, and the M.S. and Ph.D. degrees in electrical engineering and computer sciences from the University of California, Berkeley, in 2001 and 2004, respectively.

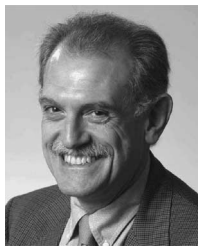
He has been with the Cadence Research Laboratories, Berkeley, since 2006, working in the System-Level Design Group. Previously, he held research positions at PARADES Research Laboratory, Rome, Italy; at the BMW Technology Office, Palo Alto, CA; at the INRIA Rhône Alpes, France; at Cadence Berkeley Laboratories; and at General Motors Research. His interests are in embedded systems design, fault-tolerant distributed systems, and control theory and applications. He has coauthored over 25 papers.

Dr. Pinello was the corecipient of two best paper awards at Design Automation Conference 2007 and Real-Time and Embedded Technology and Applications Symposium 2007.

Luca P. Carloni (S'95–M'04) received the Laurea degree (*summa cum laude*) in electrical engineering from the Università di Bologna, Bologna, Italy, in 1995, and the M.S. and Ph.D. degrees in electrical engineering and computer sciences from the University of California, Berkeley, in 1997 and 2004, respectively.

He is currently an Assistant Professor with the Department of Computer Science, Columbia University, New York, NY. He has authored over 50 publications and is the holder of one patent. His research interests are in the area of design tools and methodologies for integrated circuits and systems, distributed embedded systems design, and design of high-performance computer systems.

Dr. Carloni received the Faculty Early Career Development (CAREER) Award from the National Science Foundation in 2006 and was selected as an Alfred P. Sloan Research Fellow in 2008. He is the recipient of the 2002 Demetri Angelakos Memorial Achievement Award "in recognition of altruistic attitude towards fellow graduate students." In 2002, one of his papers was selected for "The Best of ICCAD: A collection of the best IEEE International Conference on Computer-Aided Design papers of the past 20 years."



Alberto L. Sangiovanni-Vincentelli (M'74–SM'81–F'83) received the "Dottore in Ingegneria" (*summa cum laude*) from the Politecnico di Milano, Milano, Italy, in 1971.

He is the Buttner Chair of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. He was a Cofounder of Cadence and Synopsys, the two leading companies in the area of electronic design automation. He is the Chief Technology Adviser of Cadence. He is also a member of the board of directors of Cadence, UPEK (a company he helped spin off from STMicroelectronics), Sonics, Gradient, and Accent (an STMicroelectronics–Cadence joint venture he helped found). He was a member of the HP Strategic Technology Advisory Board and is a member of the Science and Technology Advisory Board, General Motors. He has consulted for many companies, including Bell Laboratories, IBM, Intel, United Technology, COMAU, Magneti Marelli, Pirelli, BMW, Daimler–Chrysler, Fujitsu, Kawasaki Steel, Sony, and Hitachi. He is the Founder and Scientific Director of PARADES, a European Group of Economic Interest supported by Cadence and STMicroelectronics. He is a member of the High-Level Group and of the steering committee of the EU Artemis Technology Platform. He is the author of more than 800 papers and 15 books in the area of design tools and methodologies, large-scale systems, embedded controllers, hybrid systems, and innovation.

Dr. Sangiovanni-Vincentelli has been a member of the National Academy of Engineering since 1998. In 1981, he received the Distinguished Teaching Award of the University of California, Berkeley. He received the worldwide 1995 Graduate Teaching Award of the IEEE for "inspirational teaching of graduate students." In 2002, he was the recipient of the Aristotle Award of the Semiconductor Research Corporation. In 2001, he was given the prestigious Kaufman Award of the Electronic Design Automation Council for his pioneering contributions to Electronic Design Automation (EDA).