Optimal discrete controller synthesis for the modeling of fault-tolerant distributed systems

Emil Dumitrescu, Alain Girault, Hervé Marchand, Eric Rutten

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Optimal discrete controller synthesis for the modeling of fault-tolerant distributed systems

Emil Dumitrescu — Alain Girault — Hervé Marchand — Éric Rutten

N° 6137

March 2, 2007

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6137--FR+ENG

Optimal discrete controller synthesis for the modeling of fault-tolerant distributed systems

Emil Dumitrescu*, Alain Girault †, Hervé Marchand ‡, Éric Rutten§

Thème COM — Systèmes communicants
Projets POP ART, VerTeCs

Rapport de recherche n° 6137 — March 2, 2007 — 35 pages

Abstract: Embedded systems require safe design methods based on formal methods, as well as safe execution based on fault-tolerance techniques. We propose a safe design method for safe execution systems: it uses optimal discrete controller synthesis (DCS) to generate a correct reconfiguring fault-tolerant system. The properties enforced concern consistent execution, functionality fulfillment (whatever the faults, under some failure hypothesis), and several optimizations, particularly on the execution time when going through checkpoints. We propose an algorithm for optimal DCS on bounded paths. We propose model patterns for a set of periodic tasks with checkpoints, a set of distributed, heterogeneous and fail-silent processors, and an environment model that expresses the potential fault patterns. We use synchronous models, the Sigali symbolic DCS tool and Mode Automata.

Key-words: Real-time systems, safe design, fault tolerance, optimal discrete control synthesis, synchronous systems.

* INSA Lyon, http://www.insa-lyon.fr, emil.dumitrescu@insa-lyon.fr
† INRIA Rhône-Alpes, POP ART, http://pop-art.inrialpes.fr/people/girault, Alain.Girault@inrialpes.fr
‡ IRISA/INRIA-Rennes, Vertecs, http://www.irisa.fr/prive/hmarchan, Herve.Marchand@irisa.fr
§ INRIA Rhône-Alpes, POP ART, http://pop-art.inrialpes.fr/people/rutten, Eric.Rutten@inria.fr

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 —Télécopie +33 4 76 61 52 52

Synthèse de contrôleurs discrets optimale pour la modélisation de systèmes distribués tolérants aux fautes

Résumé : Les systèmes embarqués requièrent des méthodes de conception sûres fondées sur des méthodes formelles, ainsi qu'une exécution sûre fondée sur des techniques de tolérance aux fautes. Nous proposons une méthode de conception sûre pour des sysèmes à l'exécution sûre : elle utilise la synthèse de contrôleurs discrets pour générer un système tolérant aux fautes reconfigurable correct. Les propriétés assurées concernent l'exécution consistente, le remplissage de la fonctionnalité (quelles que soient les fautes, sous une certaine hypothèse de fautes), et plusieurs optimisations, notamment sur le temps des exÃ©cutions passant par des points de reprise. Nous proposons un algorithme de synthèse de contrôleurs discrets optimale sur des chemins bornés. Nous proposons des patrons de modèles pour un ensemble de tÃ¢ches périodiques avec points de reprise, un ensemble de processeurs distribués, hétérogènes et silencieux sur défaillance, ainsi qu'un modèle de l'environnement qui exprime les patrons de fautes potentiels. Nous utilisons des modÃ¨les synchrones, l'outil de SCD symbolique Sigali et les Automates de Modes.

Mots-clés : Systèmes temps-réel, conception sûre, tolérance aux fautes, synthèse de contrôleurs discrets, programmation synchrone.

1 Motivation

The motivation of this work is to propose a methodology based on discrete controller synthesis, with optimal synthesis on bounded paths, in order to model, design, and optimize fault-tolerant distributed systems.

1.1 Safety critical embedded systems

Embedded systems account for a major part of critical applications (space, aeronautics, nuclear...) as well as public domain applications (automotive, consumer electronics...). Their main features are:

- *duality automatic-control/discrete-event*: they include control laws modeled as differential equations in sampled time, computed iteratively, and discrete event systems to sequence the control laws according to mode switches;
- *critical real-time*: unmet timing constraints may involve a system failure leading to a disaster;
- *limited resources*: they rely on limited computing power and memory because of weight and encumbrance, power consumption (autonomous vehicles or portable devices), radiation resistance (nuclear or space), or price constraints (consumer electronics);
- *distributed and heterogeneous architecture*: they are often distributed to provide enough computing power to keep computing sites close to the sensors and actuators, and to allow fault-tolerance.

1.2 Problem statement

An embedded system being intrinsically critical, it is essential to insure that it is tolerant to processor failures. This can even motivate its distribution itself. In such a case, at the very least, the loss of one computing site must not lead to the loss of the whole application.

We are interested in formal methods to model systems with guarantees on their fault-tolerance capabilities. Among the various existing formal methods, we investigate the use of *discrete controller synthesis* (DCS). The advantages of using DCS are the correctness of the resulting system and the easy modifiability of the controller (thanks to automatic tools), i.e., the possibility to study and test *several* fault-tolerance objectives or failure hypotheses on the same system model, without the need to re-design the system. Specifically, our objective is:

> *To produce automatically a controller enforcing fault-tolerance for a given distributed system.*

Fault-tolerance is the faculty to *maintain functionality of a system, whatever the faults* under some failure hypothesis. To achieve this, we will need first to model our distributed systems,

and second to express formally some fault-tolerance objective, in terms of events and states of the system.

We propose to designers a methodology for modeling a system and studying the existence of fault-tolerant solutions according to several failure hypotheses and system's configurations. When a solution is found, it can be used either as a guideline for implementation (if the model was an abstract one [12]) or for deployment with a dynamic failure reconfiguring feature (this paper).

In our approach, a system consists of a set of real-time periodic tasks placed in a *configuration* onto a set of processors. Each task has a known execution cost and quality on each processor. Upon the occurrence of a fault, one or several processors become unusable, and tasks must be placed anew in another configuration, by migrating them onto another processor, so that execution can proceed. These *reconfigurations* of the system have to be controlled according to a fault-tolerance policy, enforced by a *task manager*. The latter is specified in terms of properties concerning placement constraints, reachability of termination, and optimization of costs and qualities.

1.3 Contributions

We propose to automatically produce the task manager with DCS techniques, applied to a model of the system in all its possible configurations. This model will consist of several components, each modeled as a labeled transition system (LTS), and composed in parallel; DCS will produce a property-enforcing layer on top of the components [1]. We extend previous results [13] by considering tasks with checkpoints, and using *optimal DCS* along paths.

We design and implement an algorithm for optimal DCS on bounded paths reaching a target configuration, where we introduce the possibility of optimizing systems containing 0-cost wait states. To the best of our knowledge, this feature is not available in classical optimal synthesis approaches. Yet, it is very useful in reactive systems where some states correspond to waiting for input events.

The technical context of our work is the synchronous approach[1] for the design of reactive systems [5]. This choice is motivated by the existence of a corpus of available results (programming languages, compilers, formal tools) and technologies, which already have an industrial impact. Our method is based on synchronous models, and this influences some of our choices in the design of the LTSs and on the parallel composition, as well as in already existing DCS models and tools [22] which we extended with the optimal DCS algorithm for bounded paths.

1.4 An introductory example

In order to motivate concretely the contributions in the following, we will use the example of a task on an architecture with two processors. It is initially idle, and upon a request r

[1] `http://www.synalp.org`

goes into a ready state. From there, it can be started on either processor; the choice is given through two exclusive events: a_1 for processor 1, and a_2 for processor 2. The architecture can be heterogeneous, and the performance of the task can be variable on the different processors, in terms of computations time, energy consumption, quality of service, ... The task has two phases: A, followed by B; between the two phases, there is a checkpoint event c. Upon reception of a second event c in phase B, the task terminates. We will be modeling and controlling the configurations of this task, executing on this architecture, in reaction to faults consisting of processor failures. In this case, the task will be migrated and executed on the remaining safe processor. This can happen along the duration of the task, in phase A or in phase B. When the migration occurs after the checkpoint, the task is started in the second phase, and not from the beginning.

We want to model all these possible evolutions of a system in order to compute a controller (typically: deciding upon events a_1 and a_2) that, for all possible evolutions of the architecture, will keep the task running (i.e., not being assigned to a faulty processor) up to completion (i.e., reaching termination). For this, we build a model of all the configurations of the task, and the transitions it can make between them. We do this in terms of labeled transition systems (LTS) as shown in Figure 1.

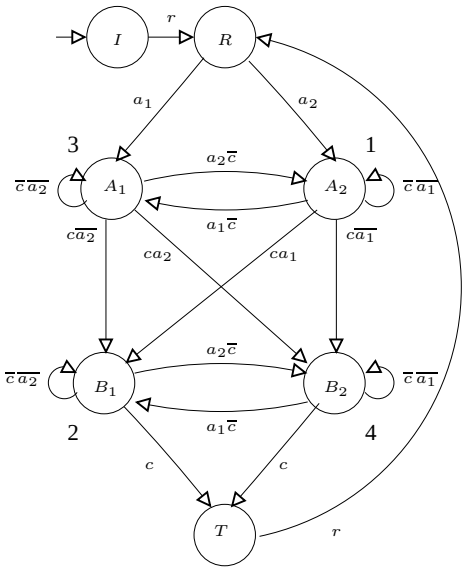


Figure 1: An introductory example.

On this very simple example, one can see that the behaviors mentioned above have been modelled. The performance aspects can be modelled as weights associated to each state. The control can then be done taking into account the cost of the paths from the ready state

to the termination state. On systems more elaborate than this one, with more complex tasks, and with a number of different tasks and a scheduler, the model obtained will feature many possible execution paths.

This paper shows our approach in:

- modelling this class of systems, by composing models of the configurations of each of the tasks, of the architecture, and the environment;
- specifying the properties they must statisfy in order to be fault-tolerant, and to optimize cost criteria;
- using optimal DCS techniques to generate automatically a controller to enforce fault tolerance.

1.5 Related work

Formal approaches to the design of fault-tolerant systems have mostly considered the problem of verification, in the context of process algebra [27, 7, 6]. They *verify* a posteriori that an existing, hand-made design (replicas interaction control, voters, etc) satisfies a certain equivalence with the nominal functionality specification, even in case of faults. In contrast, DCS approaches *synthesise* a priori automatically a controller that will insure this by construction. The principle is to consider faults as *uncontrollable* events, and fault-tolerance as the existence of behaviors able to achieve the functionality whatever the occuring faults. Planning under uncertainty is another existing approach [17], so far only demonstrated with 1-fault tolerant paths, while our DCS based approach can tolerate an arbitrary number of failures. The reachability of marked final states defines the ability to achieve functionality, and can be used as a criteria on the existence of a solution [9]; we take it as a synthesis objective.

Applications of DCS to real-time systems have been explored in various works. In the synchronous approach, with a logical, event-based time, DCS has been used to produce property enforcing layers [1] on top of a set of tasks [24]. An application of this approach to fault-tolerance [13] is preliminary to this paper.

Others are taking into account timed aspects [18], for the generation of correct application-specific schedulers, but they do not consider fault-tolerance specifically. Also, we concentrate on Boolean models; synthesis in timed or hybrid systems [2] would be more powerful, while remaining in the decidable problems, but at a very high algorithmic cost. There exist results in process algebra comparable with a form of synthesis, but that comparison is out of our scope.

Finally, in another previous work, we have used DCS for *distributed* controller synthesis, a more difficult objective that was achieved *manually* [12], whereas here we synthesise a *centralised* controller, but *automatically*.

Optimal discrete controller synthesis for discrete-event systems has been first proposed by Bellman [4] as a particular domain of dynamic programming. Marchand [15] has extended Bellman's optimality principle to tri-valued discrete systems, and has proposed and implemented several algorithms achieving optimal control synthesis according to various optimality criteria. Static criteria express simple quantitative invariants, such as bounds, over the state space of the system. They have been used to express a bound on the global usage of some ressource (memory, power-consumption, etc.).

More sophisticated dynamic techniques have also been used to find optimal strategies depending on the current state of the system and the environment reactions at a given moment.

1.6 Outline

Section 2 introduces the background of our research: fault-tolerance, DCS, optimal DCS and the generation of property-enforcing layers; it describes our technique for handling 0-cost wait states. Then, Section 3 introduces the models we propose for specifying the various parts of our systems: hardware, software, and control. In Section 4, we present in details the properties and synthesis objectives for fault-tolerance. In Section 5, we illustrate our method on an example. In Section 6, we sketch our implementation in Mode Automata and with the SIGALI tool. Finally, Section 7 concludes, discusses, and gives directions for future research.

2 Background

2.1 Discrete controller synthesis

This section gives a very brief description of DCS. As we essentially adopt an existing framework [22, 1], and propose a modeling methodology, we only introduce the useful definitions or technical aspects of the tools, and summarize the functionality.

2.1.1 Preliminaries

Labeled transition systems. Formally, an LTS is a tuple $\langle \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \delta \rangle$, where $\mathcal{Q}$ is a finite set of states, q_0 is the initial state, $\mathcal{I}$ is a finite set of input signals (produced by the environment), $\mathcal{O}$ is a finite set of output signals (issued to the environment), and δ is the transition function, i.e., a mapping from $\mathcal{Q} \times \mathcal{B}ool(\mathcal{I}) \times \mathcal{O}^* \rightarrow \times\mathcal{Q}$. Each transition has a label of the form g/a, where $g \in \mathcal{B}ool(\mathcal{I})$ must be true for the transition to be taken (g is the guard of the transition), while $a \in \mathcal{O}^*$ is a conjunction of outputs that are issued when the transition is taken (a is the action of the transition). A transition (s, g, a, s') will be graphically noted $s \overset{g/a}{\rightarrow} s'$. We use this level of definition for our modelling work, in a graphical form in the Figures of this paper.

A *path* is a sequence (possibly infinite) of transitions starting from the initial state q_0.

A *trace* is a sequence (possibly infinite) of labels associated to a path. The language of a LTS is the set of traces it can generate.

We only focus on LTSs which are *deterministic* and *reactive* :

- determinism guarantees that the system always reacts in the same manner to the same sequence of input events;
- reactivity guarantees sensitivity to any event feed from its environment.

Two LTSs $\langle \mathcal{Q}_1, q_{01}, \mathcal{I}_1, \mathcal{O}_1, \delta_1 \rangle$ and $\langle \mathcal{Q}_2, q_{02}, \mathcal{I}_2, \mathcal{O}_2, \delta_2 \rangle$ are said to be *compatible* only if their output sets are disjoint $\mathcal{O}_1 \cap \mathcal{O}_2 = \emptyset$. *The synchronous product* beteen two compatible LTSs $\langle \mathcal{Q}_1, q_{01}, \mathcal{I}_1, \mathcal{O}_1, \delta_1 \rangle$ and $\langle \mathcal{Q}_2, q_{02}, \mathcal{I}_2, \mathcal{O}_2, \delta_2 \rangle$ is the LTS $\langle \mathcal{Q}_1 \times \mathcal{Q}_2, (q_{01}, q_{02}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \delta_1 \times \delta_2 \rangle$.

2.1.2 Discrete controller synthesis on transition systems

DCS emerged in the 80's [25], with foundations in language theory. Its purpose is, given two languages $\mathcal{P}$ and $\mathcal{D}$, to obtain a third language $\mathcal{C}$ such that $\mathcal{P} \cap \mathcal{C} \subseteq \mathcal{D}$. This is a kind of inversion problem, since one wants to find $\mathcal{C}$ from $\mathcal{D}$ and $\mathcal{P}$. Here, $\mathcal{P}$ is called the *plant*, $\mathcal{D}$ the *desired system* or *objective*, and $\mathcal{C}$ the *controller*. Several teams proposed extensions and applications of this language theory technique to LTS.

In our approach, $\mathcal{P}$ is specified as a LTS, and $\mathcal{D}$ is an objective to be satisfied by the controlled system, typically *making a subset of states invariant* in the controlled system, or *keeping it always reachable*. The controller $\mathcal{C}$ obtained with DCS is a constraint restricting the transitions of $\mathcal{P}$, i.e., inhibiting those that would jeopardize the objective. The key point is that the set of inputs $\mathcal{I}$ is partitioned into two subsets, $\mathcal{I}_c$ and $\mathcal{I}_u$, respectively the set of *controllable* and *uncontrollable* inputs. The principle of DCS is that the controller $\mathcal{C}$ can only inhibit those transitions of $\mathcal{P}$ for which the guard contains at least one controllable signal, i.e., in $\mathcal{I}_c$.

As illustrated in Figure 2, the objective is expressed in terms of the system's outputs. The controller is obtained automatically from a LTS and an *objective* both specified by the user, as computed by appropriate algorithms [22] which we will use without describing them in detail here. Its purpose is precisely to act on the controllable inputs in order to achieve the objective.

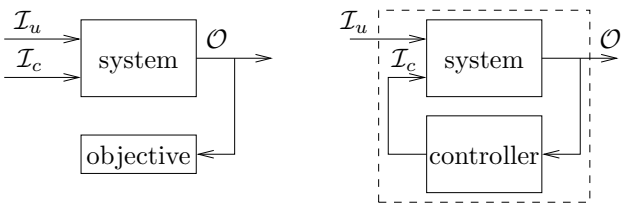


Figure 2: From uncontrolled system (left) to closed-loop control (right).

It must be noted that the order in which synthesis operations are applied does matter: indeed, their composition is *not commutative*. Reachability can not be considered before an invariance constraint, because the latter might compromise the former by removing paths and breaking reachability. On the contrary, considering reachability after invariance does not jeopardize the invariance, as it will not result in paths going out of the invariant set. Optimization should be considered last, as a means of choosing one optimal solution among the correct ones; used after reachability, it only keeps a subset of paths, which still satisfy reachability.

The result of the synthesis is a constraint, which is not readable by itself or usable "brainually"[2] by the designer, but is meant to be coupled with the system as in Figure 2, or more precisely composed as shown in Figure 12. An implementation of this coupling is proposed in Section 6.

2.1.3 One-step optimal discrete controller synthesis

It is also possible to consider *weights* assigned to the states and/or inputs/outputs of $\mathcal{P}$, and to specify that some upper or lower bound must never be reached. *Optimal controller synthesis* [21] can then be used to control transitions so as to minimize/maximize, in one step, some function w.r.t. these weights; i.e., *go only to next states with optimal weight*. There can be several equally weighted solutions, so optimization does not necessarily lead to determinism. It can be noted that this gives us only a *one step* choice i.e., a local optimal, not a global optimal on all the behaviors. With respect to our problem, such weights can model the worst-case execution time (WCET) of a given task onto a given processor, its power consumption, the amount of processor load it requires, or the quality of its results when executed on this particular processor.

In terms of the concrete transition systems seen previously, we define $C(q)$ be a cost function mapping each potential state of an LTS to a strictly positive integer cost value: $C : \mathcal{Q} \rightarrow N$. We can have several different cost functions over the state space. When composing LTSs, the value of a function is defined on the resulting global state as the sum of the local costs.

For paths across states, we can also define a path cost. The *execution cost* of a path of length k starting at state q_1, $E(q_1) = (q_1, \ldots, q_k)$ is obtained by adding the static execution costs of the states in E.

2.1.4 Optimal discrete controller synthesis on paths

Cost of the optimal strategy. Bellman's algorithm for dynamic programming computes an optimal strategy for reaching a *target* state of an LTS in presence of uncontrollable events, belonging to an adversary environment [4, 15]. By driving the uncontrollable inputs adequately, the adversary tries to prevent the LTS from reaching the target. The optimal

[2]We gratefully thank Albert Benveniste for this neologism, formed from the two words "brain" and "manually". It intends to capture the fact that such work generally involves more the "brain" than the "hands".

strategy, if it exists, is the control solution that drives the LTS towards the target state at a best cost despite the worst moves of the adversary. This computation takes two steps:

1. Computation of the best cost to reach a target state in F. This step maps each state σ of the transition system to the best execution cost achievable to reach F, by taking into account the *worst-case* moves of the adversary. This cost value is not necessarily the minimal execution cost achieable. If such an execution path does not exist, then the best cost achievable is equal to $+\infty$. Let $W : \mathcal{Q} \rightarrow N$ be the mapping function. W is defined as the greatest fixed point of the following recurrent equations:

$$W^0(q) = \begin{cases} 0 & \text{iff } q \in F \\ +\infty & \text{otherwise} \end{cases}$$

$$W^i(q) = \min \begin{cases} W^{i-1}(q) \\ \forall i > 0, \max_{\mathcal{I}_u} \min_{\mathcal{I}_c} C(q) + W^{i-1}(f(q, i_u, i_c)) \end{cases}$$

2. Use of W to generate the best trajectory reaching F. For any state q, compute the best immediate sucessor q' such that

$$W(q') = \min\{W(f(q, i_u, i_c)\} \text{ and } \forall i_u, \exists i_c : q' = f(q, x, c)$$

in a way similar to what was said in section 2.1.3.

According to Bellman, this algorithm is guaranteed to find a greatest fixed point corresponding to the optimal solution of the synthesis problem. The actual implementation of this algorithm is presented in [23].

Dealing with self-loops in the transition system. Our work considers reactive systems, especially synchronous ones, and their models in terms of LTS. A particular aspect of such models is that they explicitely represent the waiting for an event by a self-looping transition on the current state, labelled by the absence of that event, as shown in Figure 1. In this way of modelling reactive systems, the corresponding LTSs always feature cycles in the graph. Hence, Bellman's algorithm, if applied as such, will give infinite cost to most paths.

However, we know that these cycles have a particular meaning, which is the absence of action or change, and that this infinite cost does not correspond to our reactive interpretation. Our definition of reactive systems is related to the synchronous approach [5], and it states that transitions are defined for all possible valuations of the inputs [1], this including the aforementioned waiting transition. This corresponds to a time-triggered interpretation, which also accounts for the possible simultaneity of inputs. It is different from an event-triggered interpretation, where transitions are taken upon occurrence of an event: these do not feature an explicit representation of the waiting. In case of a single LTS, such an infinite cost could be avoided by having self-loops only on null-cost states. However, when having

several transition systems put in parallel, using the synchronous composition, and additive costs on resulting global states, there can be self-loops on non-null cost states. But these self-loops all result from the original trivial loops, and they all correspond to the state of the global system not changing in any of its parallel components, not making any action, and hence not involving any cost.

Therefore, we want to have a special treatment for them, and to define the cost on paths differently. We propose a modification of Bellman's algorithm in order to suit our interpretation. A distinction will be made, excluding transitions from q to q' where $q \neq q'$.

In the following sections, we will comment along the description of our models in which cases this optimization technique is applicable, and what kinds of cycles can prevent from using it.

A modified optimal path algorithm. As mentioned, Bellman's algorithm marks all uncontrollable loops with an infinite best execution cost. We argue that the particular case of 0-length loops (self-loops) deserves special processing. Uncontrollable self-loops are a common artifact for modeling the waiting for an event occurrence, which is one of the basic mechanisms in reactive systems. Indefinite waiting for an event occurrence should not penalize the best execution cost of an execution trace.

To overcome this problem, we restrict Bellman's algorithm so that self-loops only count once in the computation of the execution cost of E_l. This is done by considering in the fix-point computation that the target and source states have to be different (i.e., $q \neq q'$ in the above equation).

$$W^0(q) = \begin{cases} 0 & \text{iff } q \in Q_f \\ +\infty & \text{otherwise} \end{cases}$$

$$W^i(q) = \min \begin{cases} W^{i-1}(q) \\ \max_{\mathcal{I}_u} \min_{\mathcal{I}_c} C(q) + W^{i-1}(\delta(q, i_u, i_c)) \\ \qquad \forall q' = \delta(q, i_u, i_c) \text{ s. t } . q \neq q' \end{cases}$$

Examples of how this algorithm is actually working and the differences between the classical and adapted Bellman's algorithm will be presented further in the paper, particularly in Section 5.4.2.

2.2 Property-enforcing layers

Our modeling approach, and the way we will apply DCS, follows a framework for the automatic generation of property-enforcing layers, in a mixed imperative/declarative style [1].

A system is designed as a set of *local components*, each modeled by a LTS describing its relevant control states and transitions, and local constraints w.r.t. the environment or other components. Particularly, they feature inputs enabling the control of choices between configurations. The synchronous product of these LTSs gives a global model of the system which is a first approximation of the set of constraints that should be respected. *Global constraints* involving several components are expressed as logic properties of this product.

In the absence of a management of these global constraints, they are not satisfied; in other words, the product models the behaviors of the *uncontrolled global system*. We use general DCS techniques and tools, as presented above, in order to automatically compute and generate a *property-enforcing layer*, which, when combined with the set of communicating parallel automata, will guarantee the satisfaction of the global constraint. This controller will give values to the controllable inputs of components so that remaining behaviors are correct, *whatever* the values of the other inputs.

Advantages of this method are twofold: on the one hand, the property-enforcing layer is correct, because of the fact that it is the result of an exact computation. On the other hand, the automated nature of the process makes for an easy modifiability of designs, be it in the components behaviors or in the declarative properties; hence, a variety of global constraints can be experimented for a given system under study, providing for effective support in the design space exploration.

2.3 Fault-tolerance

Fault-tolerance has been extensively studied in the literature: [19] gives an exhaustive list of the basic concepts and terminology, [26] gives a short survey and taxonomy for fault-tolerance and real-time systems, and [16] treat in details the special case of fault-tolerance in distributed systems.

The three basic notions are *fault*, *failure*, and *error*: a *fault* is a defect or flaw that occurs in some hardware or software component; an *error* is a manifestation of a fault; a *failure* is a departure of a system from the service required. A failure in a sub-system may be seen as a fault in the global system. Hence the following causal relationship:

$$\cdots \longrightarrow \text{fault} \xrightarrow{activation} \text{error} \xrightarrow{propagation} \text{failure} \xrightarrow{causality} \text{fault} \longrightarrow \cdots$$

We assume the following *failure hypothesis*: only the processors can fail, with a *fail-silent* model. That is, a processor is either active and works fine, or faulty and does not produce any output. To tolerate such faults, we are going to make use of the *intrinsic* hardware redundancy offered by the distributed architecture: i.e., we do not wish to add extra processors but to use only the existing ones. Our goal is to apply *error treatment* techniques, such that whenever a processor will fail, the tasks that were active on it will be dynamically restarted on some other non faulty processor. The new configuration of the system reached after such an error treatment is degraded in the sense that less processors are now available, but the functionality is maintained since all the tasks are still being executed.

2.4 DCS for fault-tolerance properties enforcing

In this paper, the general framework of property-enforcing layer generation is applied specifically to fault-tolerance.

The components are tasks and processors, for which local models represent the control between configurations and the failures. In terms of Section 2.1.2, the state space of the

transition system describes configurations and error states, transitions represent startings, checkpoints, and switches between configurations.

Global constraints specify the fault-tolerance as execution coherence and the maintaining of functionality, in terms of invariance and reachability. The synthesized controller manages tasks reconfigurations in order to enforce the required fault-tolerance policy. Optimal controller synthesis refines choices between admissible control paths.

3 Abstract model of a distributed system

In this section, we specify our abstract model, and failure hypothesis. All the while, we keep in mind our objective, so as to make these abstract models suitable for DCS. So, we consider real-time systems composed of:

- a distributed heterogeneous architecture, consisting of a set of fail-silent processors, fully connected by point-to-point communication links,
- a set of periodic tasks, with the possibility to run them on the different processors, with varying characteristics (quality, power, or execution time cost),
- an application, invoking the tasks, which can be considered simply as a task management layer or as a scheduler, enforcing precedence constraints between the tasks.

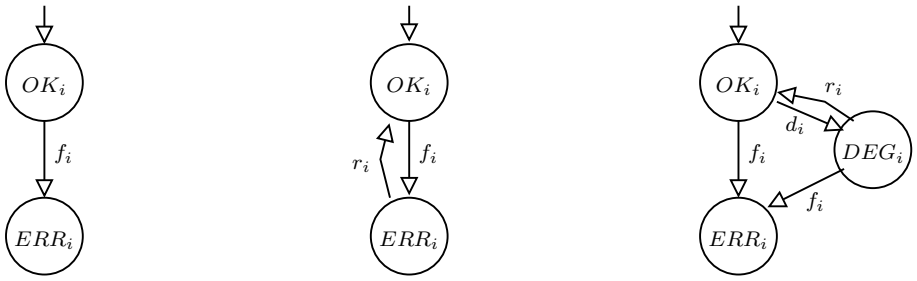
The real-time aspect of such systems comes from the execution time costs of the periodic tasks. The time cost of each task is measured thanks to a WCET analysis. Then, each task being periodic, we consider that, when executing on a processor, it uses some CPU load, computed by dividing its WCET by its period. Enforcing real-time constraints amounts thus to assigning to each processor i a CPU load *maximal bound* b_i, which should never be overtaken.

3.1 Architecture model

3.1.1 Local processor model

Each processor is modeled by the LTS of Figure 3(a), where OK_i means that the processor i is running fine, while ERR_i means that it has crashed. We assume that only the processors can fail, with a fail-silent model. Recent studies on modern processors have shown that a fail-silent behavior can be achieved at a reasonable cost [3]. Failures are also permanent, hence a processor cannot go back from the ERR to the OK state.

To model intermittent failures, we would just need to add such a "repair" transition, as in Figure 3(b). For a processor with degraded modes (e.g. at a slower speed, overloaded), we can have a model as in Figure 3(c), where upon d_i the degraded mode is entered, and upon r_i the OK_i mode is resumed. The models with intermittent failures introduce cycles in the processor model. However, it must be kept in mind that these models will be composed



(a) Permanent failure (b) Intermittent failure (c) Degraded mode

Figure 3: Processor models.

with a fault model, describing whether failures are considered only in a bounded number. In such a case, the resulting global LTS will not feature a cycle.

Processors can be used by tasks in a time-sharing manner, so that several tasks can be active on the same processor at the same time. Related to this, one might consider *exclusions between tasks*, forbidden to share the same processor because of the use of some exclusive resource. This corresponds to mechanisms of critical sections, with semaphor operations P and V enforced by the controllables. Also, related to the weights and particularly costs in power and load, individual tasks weights are to be additive: on a given processor, the global load is the sum of that of all the active tasks.

3.1.2 Heterogeneous architecture model

The processors are embedded inside a fully connected network of point-to-point communication links, like the one presented in Figure 4. We note $\mathcal{S}$ the set of all these processors. We assume that the communication links cannot fail. One processor is dedicated to executing the controller, P_0, and only the other processors are available for executing the system's tasks. Each processor must detect in real-time the other processors' failures. This can be easily implemented by making all the processors in $\mathcal{S}$ send an "I am alive" message to each other at periodic interval, like in group membership protocols [14].

The model consists of the composition of all LTSs as above. In the example, we have three of them, one for each of the processors P_1, P_2, and P_3, for which capacity bounds b_i w.r.t. power consumption are, respectively, 5, 3, and 6.

This distributed architecture is *heterogeneous*, meaning that the WCET and power consumption of each task is not the same on each processor. There may be tasks that cannot run on some processor, for instance because they require a specific hardware device (input sensor, dedicated co-processor...).

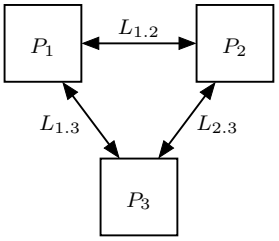


Figure 4: Distributed architecture.

3.1.3 Environment or fault model

We now need to model what failures can occur in the system. For instance, how many failures can occur? Or more precisely how many failures do we want the system to tolerate? Can failures occur in any order or are there known sequences or patterns? Can they occur simultaneously? In terms of our processor model of Figure 3, the question is how can the f_i events occur? It seems natural that all the f_i events be uncontrollable (i.e., $\in \mathcal{I}_u$), since a failure is a event intrinsically uncontrollable. But this would mean that there would be no constraints whatsoever on them. In particular, *all* events f_i could occur, meaning that all processors could fail at the same time. Of course, this would result in a total failure of the system, with no possibility at all to ensure the fault-tolerance of the system. No one expects a system to tolerate a failure of all the processors it is made of. Therefore, we need to specify the way the failures do occur in the patterns that we consider.

To model this, we choose to have a LTS modeling the environment. Its purpose is to issue the signals f_i from signals e_i produced by the environment. These signals e_i will be uncontrollable (i.e., $\in \mathcal{I}_u$), reflecting the fact that a failure can occur at any time, while the signals f_i will be local, i.e., neither in $\mathcal{I}_u$ nor in $\mathcal{I}_c$, and will be used only for computing the synchronous product of all the LTSs.

The environment model of Figure 5(a) allows only one failure to occur in the system, while the one of Figure 5(b) allows two failures to occur, possibly simultaneously (if simultaneous occurrences of failures are forbidden, it suffices to remove the three transitions from B to $F_{1,2}$, $F_{1,3}$, and $F_{2,3}$). In both cases, B is the initial state while the state $F_{i,j,k\ldots}$ records the occurrences, not necessarily simultaneous, of the failures of processors $P_i, P_j, P_k \ldots$

As a variant, according to the available knowledge about the system, one can directly specify the *failure patterns* by giving directly the LTS producing the local signals f_i from the input signals e_i. This is more expressive than specifying the number of processors that can fail. For example, Figure 5(c) corresponds to the failure pattern where up to two processors can fail, except that processors P_1 and P_3 cannot fail together, and that P_2 can fail after P_3 but P_3 can not fail after P_2.

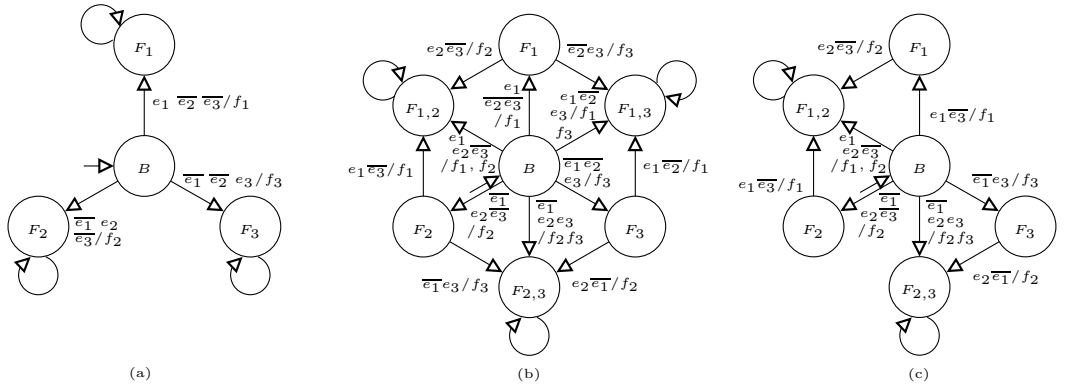


Figure 5: Fault model: (a) only one failure; (b) one or two; (c) failure pattern.

Providing such an environement model is part of the design work. The choice will depend on the knowledge of the system and the related failure assumptions. For instance, if it is unlikely for two failures to occur simultaneously, the three transitions from B to $F_{i,j}$ will be removed from the LTS of Figure 5(a).

For convenience, we introduce a failure event that signals the occurrence of at least one failure: $f \stackrel{def}{=} \bigvee_i f_i$.

As we noted earlier when presenting the processor models, and as will be seen further, when the LTS of the fault model is acyclic, then by composition with the other parts of the model it will result in an acyclic global LTS. This corresponds to the intuitively satisfying fact that we want to design systems tolerating bounded failure patterns. The technique of optimal discrete controller synthesis on paths is then applicable, and can enforce guarantees on the performance of the fault-tolerant system.

It can be noted however that we do not exclude considering applications where unbounded failure patterns have to be taken into account. In such cases, the failure model can describe what kinds of infinite sequences have to be tolerated. No fault tolerance insuring optimality or boundedness of costs on such paths can then be obtained of course, but still, objectives such as invariance, bounding of state cost, or one-step optimization remain available to the designer.

3.2 Task model

3.2.1 Simple tasks

Basic control structure pattern Each task j is formally modeled by a LTS, which describes how the control of the activity of the task is done in reaction to events.

Figure 6 shows a task model, drawn assuming that the task can be executed on the three processors of the architecture of Figure 4. It features an initial *idle* state I^j, a *ready* state

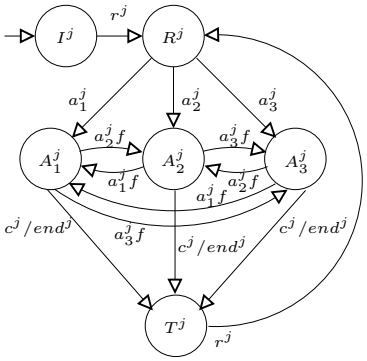


Figure 6: Task control model.

R^j after reception of the *request* signal r^j, a *terminal* state T^j, and several *active* states A_i^j, representing task *configurations*, one for each processor in the system. Here, i indicates which processor the task j is active on; since our architecture has three processors, each task LTS has three active states. By convention, subscripts/superscripts refer to processors/tasks. In the state A_i^j, task j is periodically executed on processor i, until the occurrence of the control event c^j: this is what we mean by periodic tasks. Such periodic tasks can be directly and easily modeled by Mode Automata [20].

Implicitly, each state has an additional self-loop labeled with the complement guard w.r.t. all its other outgoing transitions. For instance, state I^j has a self-loop labeled with $\overline{r^j}$, which enables the LTS to remain inside I^j until the occurrence of the signal r^j.

A transition from state A_i^j to state A_k^j represents the *re-configuration* of the system, by *stopping* task j on processor i and *restarting* it onto processor k. We call this operation a *migration*. They will be decided in order to maintain the system in a global configuration such that it keeps offering its nominal service. In particular, a migration can be decided as a reaction to a processor failure (in which case the task does not need to be stopped of course). It could also serve to balance the load between several active processors, or to comply to the energy consumption bound of a processor.

Cycles of such re-configurations could make sense, especially w.r.t. load balancing issues, but as such they would introduce cycles in the global LTS modelling the system, and this might prevent from using the application of optimal DCS on paths. In the framework of this paper, we consider reconfigurations only upon failure; this can be obtained by conditioning all reconfiguration transitions with the f event introduced earlier in the fault model: $a_i^j \wedge f$. Hence, if the fault model is acyclic, then the optimization on paths is applicable.

Another potential source of cycles is in the transition from T^j to R^j. We will see next that tasks can be called by an applications controller or scheduler: if the latter is acyclic, then so will the global LTS be.

To summarize, the basic control structure of tasks consists of a repetition of the basic pattern illustrated in Figure 7. When in a state A_2^j, where it arrived upon activation event

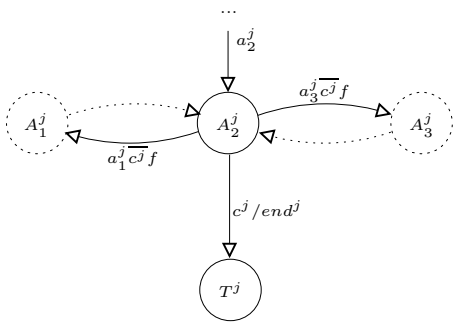


Figure 7: Task control model: basic pattern.

a_2^j (possibly conditioned by f), it is performing its associated computation on processor 2. From there, it can either:

- be migrated to another processor k upon event a_k^j (possibly conditioned by f),
- receive a control event c^j causing it to proceed in sequence, here towards terminal state T^j, with emission of an event end^j.

We will see next how this simple pattern can be extended.

In terms of controller synthesis, the signals r^j, c^j and f will be uncontrollable (i.e., $\in \mathcal{I}_u$), while the signals a_i^j will be controllable (i.e., $\in \mathcal{I}_c$).

Quantitative characteristics Some interesting characteristics can be modeled as weights associated with states [24]; we consider just simple mappings from states to integers.

Execution time is the CPU load required by each task, as measured by a WCET analysis. When a task migrates, it rolls-back to its latest checkpoint. Hence its new processor must fully accept the CPU load of the restarted task's phase.

Power consumption C_i^j of a task j is given relatively to each processor i. It is related to the WCET, but not in a linear way [8]. For our example, the values of C_i^j are given in Table 1, along with each bound b_i, which is the maximum consumption admissible by the processor i.

Quality Q_i^j of a task j is given relatively to each processor i. It can account, e.g., for the accuracy of the results produced either by a numerical computation according to the presence of special co-processors, or by different versions of an algorithm of varying depth in a heuristic search, or by an image processing operation. For our example, the values of Q_i^j are given in Table 1.

3.2.2 Tasks with checkpoints

Control structure pattern with checkpoints. Compared with tasks modeled previously, we introduce a notion of phases and checkpoint. Going from one phase A_2^j to the

		power consumption C			quality Q		
		processor			processor		
		P_1	P_2	P_3	P_1	P_2	P_3
task	T^1	4	4	2	3	5	3
	T^2	2	2	3	2	2	5
	T^3	2	3	4	2	2	5
	bound b	5	3	6			

Table 1: Consumption C_i^j, quality Q_i^j of tasks T^j on processors P_i, with bound b_i.

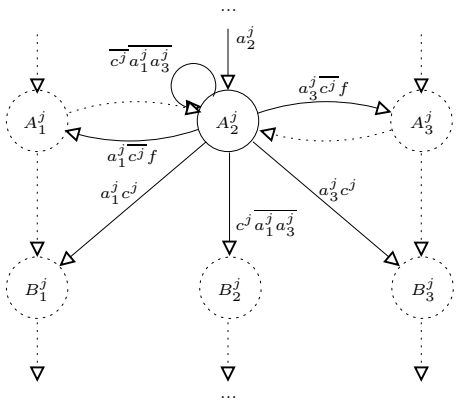


Figure 8: Task control model, with check points: simplified basic pattern.

next in sequence B_2^j is acknowledged with an uncontrollable checkpoint event c^j. The last checkpoint is actually the termination. When a task is migrated, it is restarted from the beginning of the *current phase* and not from the very start of the task. In that sense, each phase transition is a control checkpoint, and when a task is migrated it rolls back to the latest checkpoint. Hence, the task model is modified accordingly, as in Figure 8, showing a simplified extract for readability purposes, where we consider that the controllable events a_i^j are exclusive. From an active state, one can make a transition towards activation on another processor in case of migration with a_k^j, or towards the next phase in sequence upon c^j in the absence of migration, or when migration and checkpoint occur simultaneously, towards the next phase on another processor.

As before, controlled re-configurations introduce cycles in the task model, but if they are conditioned by fault occurrences f and the fault model is acyclic, then there will be no cycle in the global LTS, and optimal discrete controller synthesis on paths is then applicable. Reconfigurations can be performed freely at the same time as a checkpoint, without jeopardizing this.

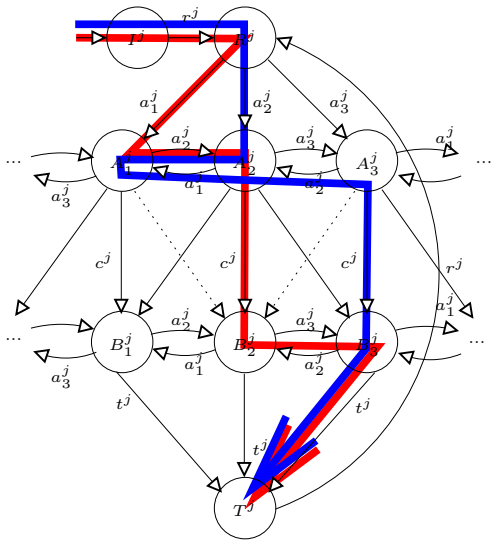


Figure 9: Task control model, with check points: example.

Complete task example with checkpoints. We assume as before that the task can be executed on the three processors of the considered architecture. The difference is that configurations are represented by different states A_i^j and B_i^j for each phase (in our example there are two phases) of task j on processor i. Figure 9 shows, as before, an initial idle state I^j, in which a request r^j causes a transition to the ready state R^j. There the choice is given to the controller to go to either of the three processors and start the computation accordingly, which is represented by state A_i^j. The cost is considered to be "consumed" right away: it is the worst case execution time, and the worst case here is that a migration occurs just before the end of the computation. Figure 9 is built by repetition of the pattern of Figure 8, and addition of the idle, ready and terminated states; some transitions have been omitted or dotted for readability. For claritiy, transitions going to "..." are meant to go to the mentioned state at the other side of the Figure.

Thick arrows in Figure 9 show two possible scenarios of the execution, with transitions corresponding to migrations and to phase transition and checkpointing.

One of them (the darker one, of magenta color) start on processor 1, in state A_1^j. Then processor 1 fails immediately, and the computation is migrated to processor 2 : the cost $\mathcal{C}(A_1^j)$ will be augmented by $\mathcal{C}(A_2^j)$. Computation proceeds to A'^j_2, and on the occurrence of chk^j to the next phase, on the same processor, in state B_2^j, where the cost will be augmented by $\mathcal{C}(B_2^j)$, and then B'^j_2. Later, before the end of the phase, processor 2 fails, and a migration occurs to processor 3, with a restarting at the last checkpoint, i.e. in state B_3^j, with cost $\mathcal{C}(B_3^j)$. From there computation proceeds to B'^j_3, and finally terminates with t^j

to T^j. Another example of scenario (in lighter, cyan color) is that computation is started on processor 2: A_2^j, which fails after a while, before chk^j, leading to processor 1: A_1^j, which also fails, immediately, leading to processor 3: A_3^j, from where it proceeds to the termination.

Along those paths, the time spent in computation is the cumulation (additive) of times in each state, given that the worst case is that the migration occured just before terminating the phase. On these bases, controlling the migrations in order to bound or optimize a traversal time can be posed as an optimal synthesis problem on these paths. A quantitative example is detailed in Section 5.4.2.

3.3 Application model

An application is made of the invocation of a set of tasks, considering the latter either as a task server receiving requests, or including a scheduler or program.

3.3.1 Tasks server

If the system consists of n tasks, there will be n corresponding LTSs in parallel. Their synchronous composition as in Mode Automata [1, 20] represents all behaviors, i.e., all possible configurations, in response to all possible sequences of requests and termination events.

The composition of quantitative characteristics is considered, in this paper, to be additive. It is clear for CPU loads or power consumption on each processor P_i, where we have for tasks j: $C_i = \sum_j C_i^j$.

Regarding quality, we consider overall quality to be the means of that of active tasks, which can be understood in the same way as papers submitted for a conference receive a global mark that is the means of the various markings. We will use quality just to choose the transitions towards the next states with the highest quality; hence, we do not need to divide and can just use the sum of qualities, for processors i and tasks j: $Q = \sum_i \sum_j Q_i^j$.

As we noted earlier, in such a system, there is a cycle on the starting of tasks, hence paths are infinte, and optimal DCS on paths makes no sense and is not applicable; however the other proposed synthesis objectives are meaningful.

3.3.2 Scheduler or program

A scheduler or program can be in charge of emitting the task requests in a given sequence. Its purpose is to schedule the tasks according to the precedence graph specified by the user: it must issue the signals r^j in the correct order, so that the tasks become ready (in the R^j state) in such a way that the precedence constraints are satisfied.

If we consider the example of Figure 10, the scheduler first issues r^1, then after receiving end^1, it issues r^2 and r^3, therefore executing T^2 and T^3 in parallel, and finally, once it has received end^2 and end^3, it issues r^1. The transition system is shown in Figure 11: it features a terminal state T. This model shows where it is possible to have a set of finite paths from

the inititial state to the terminal state, and this is where optimal DCS on paths can be applied.

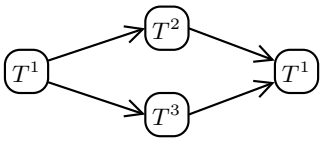


Figure 10: Precedence constraints.

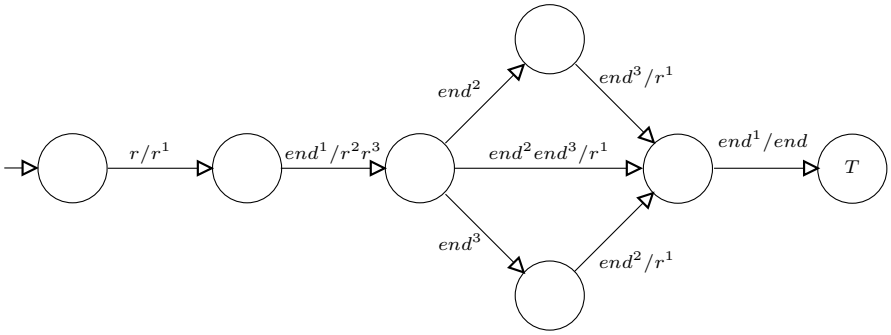


Figure 11: Precedence constraints: transition system.

Note that such schedulers or programs can be obtained from higher-level, domain-specific languages [11, 10].

3.4 System model

Finally, the model of the multi-processor, multi-task system is built by composing the different local models introduced previously: one for the environment model, one for each processor, one for each task, one for the scheduler, and one for the controller. This is illustrated in Figure 12 for a complete system made of 3 processors, 2 simple tasks and 1 task with 2 phases.

Scheduling (deciding *in which order* tasks are executed) and distribution (deciding *where* they are executed) are decoupled here: the scheduler schedules the tasks according to the precedence constraints, while the controller dynamically distributes the tasks according to the fault-tolerance policy.

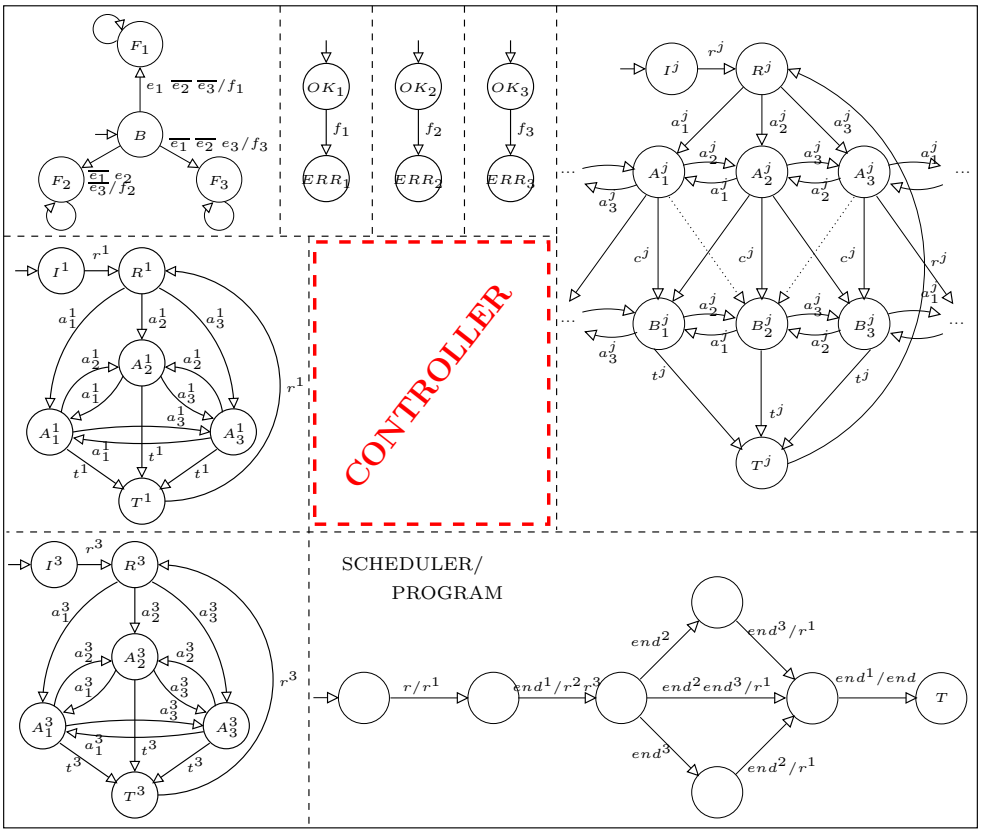


Figure 12: A complete system with 3 processors and 3 tasks, and a controller.

4 Properties, objectives and fault-tolerance

The fault-tolerance policy is specified declaratively by a set of properties and objectives. The fault-tolerance specificity of these properties is twofold. On the one hand, they are meant to be considered upon models as described above, where all faults, recoveries or failures behaviors are represented. On the other hand, they characterize failed states (e.g., consistent placement constraints characterize states where the system is not viable), as well as the tolerance, meaning the notion of fulfilling functionality whatever the faults.

4.1 Properties

4.1.1 Insuring consistent execution

Property 1 (No task is active on a failed processor)
$\neg \bigvee_j \bigvee_i (A_i^j \wedge Err_i)$.

Property 1 is contradicted whenever a task T^j is active on processor P_i (i.e., in state A_i^j) while P_i is in Err_i. The synthesis objective is to make it *invariantly true*. If the system, as modeled by the designer, is such that in each state there exists a transition to a safe state (i.e., one where Property 1 holds), then the synthesis will succeed and the controlled system will always be able to react to a processor failure by moving to a safe state. Otherwise the synthesis will fail, indicating to the designer that her/his system cannot be made fault-tolerant.

Property 2 (Tasks active are within processor capacity)
$\forall i, C_i \leq b_i$.

Property 2 is contradicted whenever the cumulated cost of all tasks active on a given processor exceeds its capacity bound. Again, the synthesis objective is to make it *invariantly true*. Typically, this objective can have the effect of inhibiting the transition from R^j to any active state A_i^j for a task j, if taking this transition means that a later processor failure, specified in the environment model, will *not* be tolerated without bounding problems. Here, the DCS computes the most permissive controller such that *all* failures are guaranteed to be tolerated without bounding problems. A terminating task can then release another waiting task.

4.1.2 Insuring functionality

The previous properties were just simple state properties, used to avoid inconsistent configurations. The discrete controller can inhibit indefinitely the start of a task if there is a possibility that the only remaining processor has too low a bound for it (after the other ones have failed). In that case, there is no solution for insuring functionality, defined here as reaching termination. In other terms, tasks are activated only when *"the path is clear and wide enough all the way down"* to termination, even in case of failures.

Property 3 (The functionality is fulfilled) *From all reachable states, the terminal state T of the program is reachable.*

For the case of a tasks server, without a scheduler or program, one should make rachable the configurations such that $\bigwedge_i T^i$.

Property 3 states that whatever the faults, as specified in the environment model, in any sequence and possible simultaneity, a terminal configuration can be reached, for any occurrences and orders of incoming task requests and terminations. This property is instrumental in characterizing fault-tolerance, as it excludes behaviors where all activity would be frozen in the waiting states in order to avoid jeopardizing Properties 1 and 2. It can serve to detect systems which do not have the capacity (logical or quantitative) to actually tolerate faults while continuing to deliver their nominal functionality.

It is different from previous properties in the sense that it considers not only the current state, but the trajectories of the system, requiring them to be able to reach termination.

4.2 Optimizing costs and qualities

4.2.1 One-step optimization

It is a matter of adopting a policy, by making switches only to the *next* configurations such that they:

- maximize the overall quality, when the quality of tasks varies according to the processor;
- minimize the global consumption, which can be defined as the sum of costs of tasks on processors.

Also, having this notion of quality (zero on inactive states and positive when active) provides for a way of imposing progress to the controlled system, where the option of remaining in the waiting state endlessly is removed; hence, proceeding to activity, and nearing to completion, is pushed forward. Another, more self-standing, way of doing things would be to have a separate weight accounting for the cost for waiting, and to minimize it [1].

4.2.2 Optimizing costs along paths through phases

It consists of choosing the sequence of phases where migrations minimize the cost between the ready state and the termination state. In the example of Figure 9, different paths across checkpoints and migrations can have different costs, and the choice of migration has to be made according to the other constraints of the application: the invariance properties must always hold, and the schedule must have the best cost that can be achieved despite the worst scenarios driven by the uncontrollable events. The optimal synthesis algorithm achieves the cost optimization through phases. It computes W_{Q_f}, the best cost function, that maps each state of M to the best execution cost achievable to reach the target $Q_f = \{(T^1, \ldots, T^n)\}$.

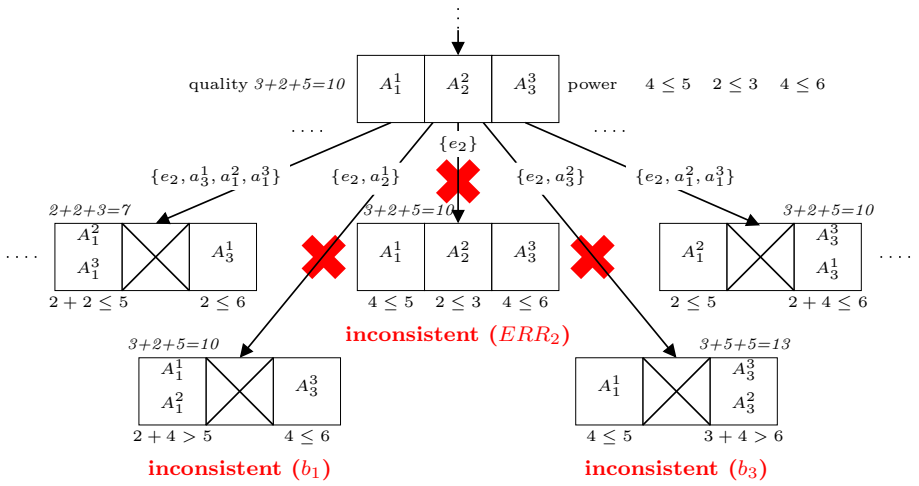


Figure 13: Example of states (only configuration is shown) and transition control.

An optimal solution exists iff $W_{Q_f}(q_0) < +\infty$. Then, the synthesis objective is expressed as follows:

$$\forall q \in Q \text{ choose } i_c^{min} \text{ s.t.:} \forall i_u, \forall i_c \neq i_c^{min} : W_{Q_f}(\delta(q, i_c^{min}, i_u)) \leq W_{Q_f}(\delta(q, i_c, i_u))$$

If an optimal solution exists ($W_{Q_f}(q_0) < +\infty$) then the reachability of the target state is guaranteed and hence, the functionality fulfillment Property 3 is also satisfied.

5 Illustrative scenarii

5.1 Property 1: consistent execution

On the basis of our running example of three tasks and three processors, we illustrated in Figure 13 an extract of the transition system. It shows configurations, with active tasks in the squares corresponding to the respective processors, and with associated weights, of quality and cost. The transitions that have to be inhibited by the controller are crossed, and the avoided configurations are labeled with the reason why.

If P_2 becomes faulty (event e_2, state ERR_2), then no task should be active on it (states A_2^1, A_2^2, and A_2^3). The same goes for P_1 and P_3. Obviously, for a fault model where all processors can fail, no controller can be found satisfying the objective: it can not start a task without risking all processors to fail before its termination, and therefore the behavior will remain stuck in the ready state for all requested tasks.

Along the same lines, tasks with placement constraints can make a system harder to control: indeed, once active on a processor P_i, there must always be another processor able to host them in case of a failure of P_i.

5.2 Property 2: bounded capacity

For the sake of the example, we consider a global configuration where we have T^1 onto P_1 ($4 \leq 5$), T^2 onto P_2 ($2 \leq 3$), and T^3 onto P_3 ($4 \leq 6$) (in the task server view of the system). Then, if P_2 crashes, T^2 is forced to migrate either onto P_1 or onto P_3. However, none of these two choices meet the constraint on the processor maximal utilization bound. Indeed, the sum of costs of T^1 and T^2 on P_1 would be $2 + 4 > 5$, while T^2 and T^3 on P_3 would give $3 + 4 > 6$. Hence, the controller forces more migrations, e.g., T^1 onto P_3 and T^2 onto P_1. This time constraints on the bounds will be met both on P_1 ($2 \leq 5$) and on P_3 ($2 + 4 \leq 6$).

A solution can be found when, after the other processors have failed as far as the environment model says, the remaining processors with the smallest capacity are still able to host all the active tasks. This constraint can also block the system in the ready states, because the path is not clear and wide enough for execution. Here, as well as for the previous objective, the environment model can have a determining influence: if it excludes intolerable fault patterns, then a solution can be found.

Also, a task model without the possibility to have the control waiting in the ready state until a favorable configuration is reached, allows less solutions. In that case, having a program or scheduler can have an impact, in that only certain subsets of tasks can be activated in parallel. This requires less capacity on the processors than a task server where the worst case is that all tasks are active in parallel. On the other hand, with tasks with a waiting state, the actual sequencing is under control of the controller, and a solution can exist, which proceeds sequentially one task after another. For such tasks, considering a program or scheduler is therefore not useful in the search of control solutions.

5.3 Property 3: functionality fulfilment

The results vary depending on the environment model:

- for a one fault model (Fig 5-a), everything works fine, as capacity is sufficient on any group of two remaining processors;
- for a two faults model (Fig 5-b), capacity of P_2 is insufficient to accommodate for task T^1, therefore no controller can insure functionality for all possible sequences of faults and requests;
- for the fault pattern example (Fig 5-c), a solution can be found, as the pathological processor configuration (P_2 only survivor) is excluded by the fault model.

One can note that, would the capacity bound of P_2 be a little higher, a solution would exist for the two faults model: changing the bounds allow us to obtain different controllability solutions. When no solution is found, the user must relax some of the system's constraints: either the environment model, or the power consumption bounds... When one solution is found, it means that we have a controller that will dynamically allocate the tasks onto the live processors, while guaranteeing that all processor failures will be tolerated and that the cumulative power consumption will always remain smaller than the bound on each processor.

5.4 Optimizations

5.4.1 One-step optimizations

This enables us to further restrain behavior, using values as in Table 1, to maximize quality (and possibly forcing migrations just to achieve this), and then to minimize the power consumption cost. As said in Section 2.1, there may be several solutions with equal weights. The example in Figure 13 shows two remaining configurations, with qualities 7 (left) and 10 (right)

These criteria can be played around with, for the same system under study: minimal consumption can be applied first, before maximizing quality in the remaining solutions.

5.4.2 Optimization on paths

Figure 14 illustrates the execution of the synthesized controller, according to the particular scenario of two tasks running on three processors, each task having two phases. The controller is generated under the assumption that both tasks start at the same moment, i.e., events r_1 and r_2 are received simultaneously. Besides, we also assume that both tasks execute only once, i.e., r_1 and r_2 are only received once. The fault model used is presented in Figure 5. Static costs are represented as integer numbers next to their corresponding states.

In this example, the best execution cost for task T^1 would be $1 + 1 + (2 + 1) + 1 = 6$, which corresponds to executing its first phase on P_3 and its second phase on P_1. The best execution cost for task T^2 is $1 + 1 + (1 + 1) + 1 = 5$, which corresponds to executing its first phase on P_2 and its second phase on P_1.

The run proceeds as follows. At the beginning of the simulation, T^1 is scheduled on P_3 and T^2 is scheduled on P_2. At that moment, processor P_2 fails. T^2 must migrate immediately and the best cost solution is offered by processor P_3. Task T^1 remains on processor P_3. The tasks can execute their own checkpoint independently of each other, when receiving the corresponding uncontrollable event $c^{1,2}$. Just after a checkpoint, processor migrations can also occur for optimality reasons: both T^1 and T^2 migrate respectively from P_3 to P_1 in order to achieve their best execution cost. Each task terminates when receiving an uncontrollable event $t^{1,2}$.

Thus, T^1 ends with a total execution cost of 6, which happens to be its lowest possible execution cost. On the other hand, task T^2 ends with a total cost of 7, instead of the minimal cost 5, due to the failure of P_2 and the migration from P_2 to P_3.

The way uncontrollable self-loops are handled is fundamental. In the example, such self loops are almost ubiquitous: each time there is an indefinite waiting for an uncontrollable event. The start events $r^{1,2}$, checkpoint events $c^{1,2}$ and termination events $t^{1,2}$ can be possibly awaited forever. The classical Bellman algorithm for optimal synthesis would directly return the worst-case cost for all these self-loops, which is $+\infty$; in this case, no optimal solution would exist and all schedules would be equivalent since they would all cost $+\infty$. However, as mentioned before, self-loops should not penalize the worst-case execution cost. Our algorithm computes the best execution cost by counting the self-loop states only once.

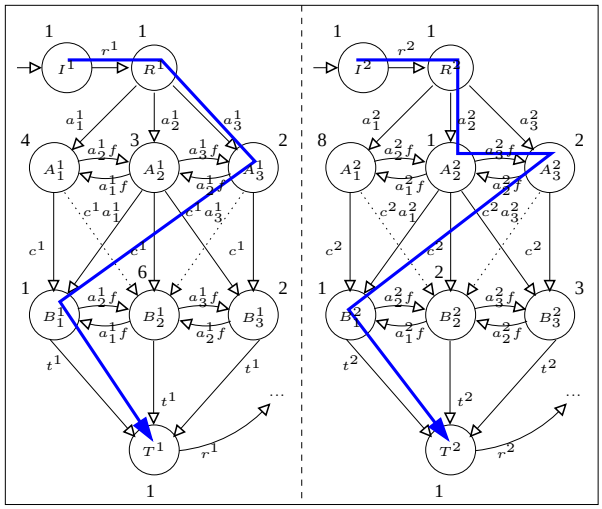


Figure 14: Simulation result for 2 tasks executing on 3 processors.

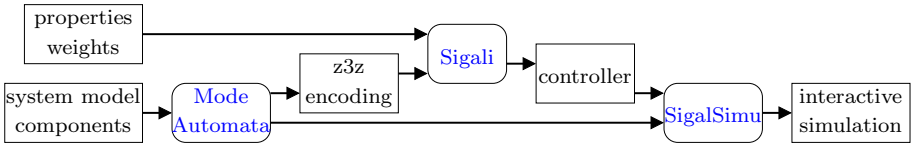


Figure 15: Tools used.

The optimal scheduler generated allows indefinite waiting for all uncontrollable events and performs on-line reconfiguration each time a failure occurs and/or the global execution cost can be improved.

6 Implementation

As we mentioned in the beginning, we are using existing synchronous and DCS techniques *as such*, and hence will not present, in this limited space, details available elsewhere. MATOU[3] [20] was used for writing the model of our systems as sets of mode automata, while the symbolic model-checker and DCS tool SIGALI[4] [22] was successfully used to automatically synthesize fault-tolerant systems from a high-level specification, and SIGALSIMU [1] was then used to co-simulate the system and the controller, as illustrated in Figure 15.

[3] `http://www-verimag.imag.fr/~maraninx/MATOU`
[4] `http://www.irisa.fr/vertecs/Logiciels/sigali.html`

We first considered simple tasks, and consistent execution objectives, and then extended our objectives with functionality fulfillment and optimization.

We then augmented the SIGALI tool with an implementation of our optimal synthesis technique for systems with self-loops.

Our method is limited by the technological state of the existing DCS tools, basically the same limitations as with model checking tools. Given the current trend in this domain (symbolic state space exploration, abstract interpretation, widening operators ...), we believe that future improvements in DCS tools will make it an efficient solution for industrial size problems.

7 Conclusion

7.1 Results

We have shown how to model a real-time distributed system, its heterogeneous architecture, and its environment in order to produce automatically a controller enforcing fault-tolerance. It reacts to the occurrences of failures by migrating tasks according to the fault-tolerance policy. For this, we have applied DCS to LTS models of the whole system, with objectives regarding consistent execution, functionality fulfillment, and optimizations.

We have coded and applied an algorithm for optimal DCS on bounded paths reaching a target configuration, where we introduced the possibility of optimizing systems containing waiting transitions, which is meaningful w.r.t. to reactive systems..

7.2 Discussion

Fault-tolerance for embedded systems can be divided into two classes of approaches: static or dynamic. In the *static* approach, task redundancies are added such that any occurrence of failures be tolerated during the execution; the drawback is that this is expensive since one has to pay the overhead of redundancy even in the absence of failures; the advantage is that a bound on the system's reaction time can be computed prior to the system's deployment, with the guarantee that this bound will hold whatever the occurrence of failures during the execution. In the *dynamic* approach, mechanisms are added to the system such that the system will be able to react dynamically to any occurrence of failures during the execution; the drawback is that no bound can be computed on the system's reaction time since this depends on the unpredictable occurrence of failures; the advantage is that no overhead has to be paid in the absence of failure; that is, until a failure occurs, the execution cost of the fault-tolerant system is (almost the same as) that of the corresponding non fault-tolerant one.

We believe that our approach is interesting in the sense that, when the DCS actually succeeds in producing a controller, we obtain a system equipped with a *dynamic* reconfiguring mechanism to handle failures (i.e., the controller), with a *static* guarantee that all specified failures will be tolerated during the execution, and with a known bound on the system's

reaction time. In other words, we have the advantages of both approaches. But remember that this is true only when the DCS succeeds. If DCS fails, this failure can concern either the invariance properties or the optimization. The failure of an invariant property means that it cannot be guaranteed that a task is never scheduled on a faulty processor: the model under development is not fault tolerant. The failure of the optimal synthesis means that there is no optimal path towards the specified target, i.e., all paths cost $+\infty$ or the target is unreachable.

7.3 Perspectives

Interesting perspectives concern:

- variants on the *model of tasks*, for instance having several modes to account for Dynamical Voltage Scaling (DVS), where a slower speed is cheaper in terms of power, or *degraded modes* for the same functionality,
- other *logical properties* of interest are exclusions between tasks, and sequencing constraints, using observers; other *quantitative properties* of interest are the use of devices (sensors, co-processors), managing memory use, bounds on migration costs, minimum levels of quality, ...

References

[1] K. Altisen, A. Clodic, F. Maraninchi, and E. Rutten. Using controller-synthesis techniques to build property-enforcing layers. In *Proceedings of the European Symposium on Programming, ESOP'03*, Warsaw, Poland, April 2003.

[2] R. Alur, O. Bournez, T. Dang, O. Maler, and A. Pnueli. Effective synthesis of switching controllers for linear systems. *Proc. of the IEEE*, 88:1011–1025, 2000.

[3] M. Baleani, A. Ferrari, L. Mangeruca, M. Peri, S. Pezzini, and A. Sangiovanni-Vincentelli. Fault-tolerant platforms for automotive safety-critical applications. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES'03*, San Jose, USA, November 2003. ACM.

[4] R. Bellman. *Dynamic programming*. Princeton University Press, 1957.

[5] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proc. of the IEEE*, 91(1):64–83, January 2003. Special issue on embedded systems.

[6] C. Bernardeschi, A. Fantechi, and L. Simoncini. Formally verifying fault tolerant system designs. *The Computer Journal*, 43(3), 2000.

[7] G. Bruns and I. Sutherland. Model checking and fault tolerance. In *Proceedings 6th International Conference on Algebraic Methodology and Software Technology, AMAST'97*, Sidney, Australia, 1997.

[8] A.P. Chandrakasan, S. Sheng, and R.W. Broderson. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, 1992.

[9] K.-H. Cho and J.-T. Lim. Synthesis of fault-tolerant supervisor for automated manufacturing systems: A case study on photolothographic process. *IEEE Trans. on Robotics and Automation*, 14(2):348–351, April 1998.

[10] Gwenaël Delaval and Eric Rutten. A domain-specific language for multi-task systems, applying discrete controller synthesisd. *Journal on Embedded Systems* (special issue on Synchronous Paradigm in Embedded Systems), *(to appear)*, 2007. `http://www.hindawi.com/journals/es/raa.84192.html` .

[11] Gwenaël Delaval and Éric Rutten. A domain-specific language for task handlers generation, applying discrete controller synthesis. In *Proc. of the 21st ACM Symp. on Applied Computing, SAC 06, Poitiers, France, April 23–27*, 2006.

[12] E. Dumitrescu, A. Girault, and E. Rutten. Validating fault-tolerant behaviors of synchronous system specifications by discrete controller synthesis. In *IFAC Workshop on Discrete Event Systems, WODES'04*, Reims, France, Sept. 2004.

[13] Alain Girault and Éric Rutten. Discrete controller synthesis for fault-tolerant distributed systems. In *Proceedings of the Ninth International Workshop on Formal Methods for Industrial Critical Systems, FMICS 04,* September 20-21, 2004, Linz, Austria, pages Electronic Notes in Theoretical Computer Science ENTCS, Volume 133 , 31 May 2005, Pages 81–100. `http://www.sciencedirect.com/science/journal/15710661`, `http://dx.doi.org/10.1016/j.entcs.2004.08.059`, 2004.

[14] R. Guerraoui and A. Schiper. Consensus service: A modular approach for building agreement protocols in distributed systems. In *26th IEEE Int. Symp. on Fault-Tolerant Computing, FTCS'96*, Sendai, Japan, June 1996.

[15] H.Marchand. *Méthodes de synthèse d'automatismes décrits par des systèmes à événéménts discrets finis*. PhD thesis, Université de Rennes-I, 1997.

[16] P. Jalote. *Fault-Tolerance in Distributed Systems*. Prentice Hall, 1994.

[17] R. Jensen. DES controller synthesis and fault tolerant control – a survey of recent advances. Res. report TR-2003-40, ITU, Copenhagen, Denmark, Dec. 2003.

[18] Ch. Kloukinas and S. Yovine. Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems. In *5th Euromicro Conference on Real-Time Systems (ECRTS'03), Porto, Portugal, July*, 2003.

[19] J.-C. Laprie et al. *Dependability: Basic Concepts and Terminology.* Dependable Computing and Fault-Tolerant Systems. Springer-Verlag, 1992.

[20] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[21] H. Marchand, O. Boivineau, and S. Lafortune. Optimal control of discrete event systems under partial observation. In *Proc. of the 40th IEEE Conf. on Decision and Control, CDC'01,* Orlando, Florida, dec, 2001.

[22] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the Signal environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4):325–346, October 2000.

[23] H. Marchand and M. Le Borgne. On the optimal control of polynomial dynamical systems over z/pz. In *4th IEE International Workshop on Discrete Event Systems*, pages 385–390, Cagliari, Italie, August 1998.

[24] H. Marchand and E. Rutten. Managing multi-mode tasks with time cost and quality levels using optimal discrete controller synthesis. In *Euromicro Conference on Real-Time Systems, ECRTS'02*, Vienna, Austria, June 2002.

[25] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, January 1987.

[26] J. Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and Systems Safety*, 43(2):189–219, 1994.

[27] H. Schepers and J. Hooman. Trace-based compositional proof theory for fault tolerant distributed systems. *Theoretical Computer Science*, 128, 1994.

Contents

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399