Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 133 (2005) 81–100
www.elsevier.com/locate/entcs

Modeling Fault-tolerant Distributed Systems for Discrete Controller Synthesis

Alain Girault, Eric Rutten[1]

INRIA Rhône-Alpes, POP ART,
655 avenue de l'Europe, 38334 Saint-Ismier cedex, FRANCE.

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 discrete controller synthesis (DCS) to generate a correct reconfiguring system. The properties enforced concern consistent execution, functionality fulfillment (whatever the faults, under some failure hypothesis), and several optimizations. We propose model patterns for a set of periodic tasks, a set of distributed, heterogeneous and fail-silent processors, and an environment model that expresses the potential fault patterns. We outline an implementation of our method, using the Sigali symbolic DCS tool and Mode Automata.

Keywords: Discrete controller synthesis, fault-tolerance, real-time systems.

1 Introduction

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;

[1] Email: Alain.Girault@inrialpes.fr, Eric.Rutten@inrialpes.fr

doi:10.1016/j.entcs.2004.08.059

- *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 and to keep computing sites close to the sensors and actuators.

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. 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 [9]) or for deployment with a dynamic failure reconfiguring feature (this paper).

In our approach, a system consists of a set of tasks placed in a *configuration* onto a set of processors. Upon occurrence of a fault, one or several processors become unusable, and tasks must be placed anew in another configuration, by restarting 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.

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].

The technical context of our work is the synchronous approach [2] for the design of reactive systems [4]. This choice is motivated by the existence of a corpus of available results (laguages, compilers, formal tools) and technologies, which already have an industrial impact. Our method is compatible with synchronous models, and this influences some of our choices in the LTSs and composition, as well as in already existing DCS, applied as such [17].

1.3 Related work

Formal approaches to the design of fault-tolerant systems have mostly considered the problem of verification, in the context of process algebra [21,6,5]. They *verify* 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 [12] *synthesise* 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 [12], so far only demonstrated with 1-fault tolerant paths. We place ourselves in the framework of reactive systems, finite state machines, and the use of synthesis of exact most permissive controllers. Moreover, we tolerate *several* failures, not only one. 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 [8]; we take it as a synthesis objective. Other works on applying DCS to real-time systems exist [13], taking into account timed aspects, 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 efficiency cost. There exist results in process algebra comparable with a form of synthesis, but that comparison is out of our scope. Finally, in another parallel work, we have used DCS for *distributed* controller synthesis, a more difficult objective that was achieved *manually* [9],

[2] http://www.synalp.org

whereas here we synthesise a *centralised* controller, but *automatically*.

2 Background

2.1 Fault-tolerance

Fault-tolerance has been extensively studied in the literature: [14] gives an exhaustive list of the basic concepts and terminology, [20] gives a short survey and taxonomy for fault-tolerance and real-time systems, and [11] 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 state 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.

2.2 Discrete controller synthesis

This section gives a very brief description of DCS. As we adopt an existing framework [17], we do not reproduce the definitions or technicalities of the tools, but just summarize the functionality. DCS emerged in the 80's [19], 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*.

Recently, several teams proposed extensions and applications of this language theory technique to labeled transition systems (LTS). Formally, an LTS

is a tuple $\langle \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} \rangle$, 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 $\mathcal{T}$ is the transition relation, i.e., a subset of $\mathcal{Q} \times \mathcal{B}ool(\mathcal{I}) \times \mathcal{O}^* \times \mathcal{Q}$. 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).

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 constrain 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 1, the objective is expressed in terms of the system's outputs and the controller is obtained automatically and its purpose is precisely to act on the controllable inputs in order to achieve the objective.

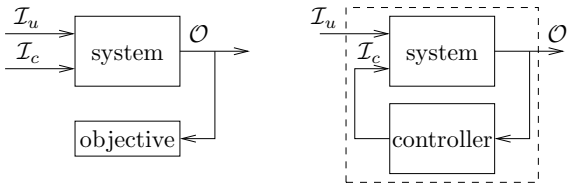

Fig. 1. From uncontrolled system (left) to closed-loop control (right).

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* [16] can then be used to control transitions so as to minimize/maximize, in one step (or on bounded paths), 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.

The order in which synthesis operations are applied does matter: indeed, their sequence 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 choice among correct solutions; even after reachability, it will keep only some paths, which should always satisfy it.

The result of the synthesis is a constraint, which, as it is computed automatically, is not quite readable or usable "brainually" by the designer, but is meant to be coupled with the system as in Figure 1, or more precisely composed as shown in Figure 6. An implementation of this coupling is proposed in Section 6.

2.3 Property-enforcing layers

Our approach follows a framework for the automatic generation of property-enforcing layers, in a mixed imperative/declarative style, based on DCS [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.

In this paper, this general framework is applied specifically to fault-tolerance. The components are tasks and processors, for which local models represent configurations and failures. 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.

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 time cost),
- an application, invoking the tasks, which can be considered simply as a task management layer, or as a scheduler or program, enforcing precedence constraints between the tasks.

The real-time aspect of such systems comes from the 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 a CPU load *maximal bound*, which should never be overtaken.

3.1 Architecture model

3.1.1 Local processor model

Each processor is modeled by the LTS of Figure 2, 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 transition.

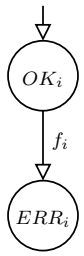

Fig. 2: Processor model.

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. 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. Each processor i has a *quantitative bound* b_i, specifying its maximum power capacity.

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 3. 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 [10].

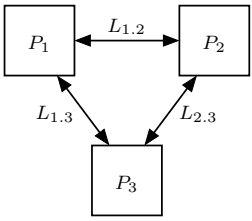

Fig. 3: Distributed architecture.

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...).

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? Can they occur simultaneously? In terms of our processor model of Figure 2, 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. 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 building the synchronous product of all the LTSs.

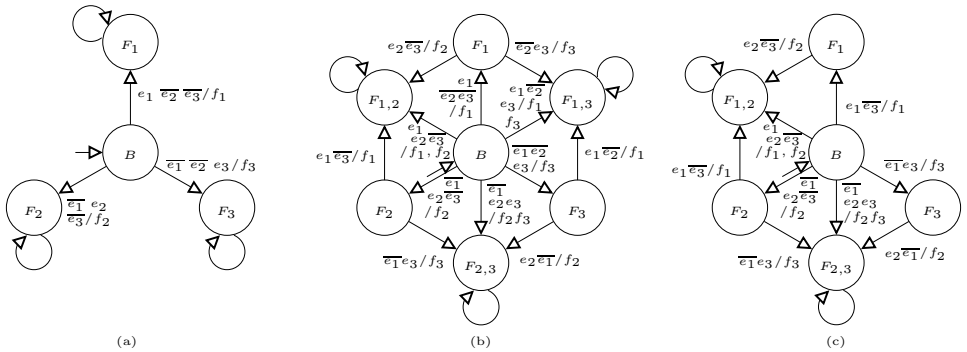

Fig. 4. Fault model: (a) only one failure; (b) one or two; (c) failure pattern.

The environment model of Figure 4(a) allows only one failure to occur in the system, while the one of Figure 4(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 4(c) corresponds to the failure pattern where up to two processors can fail, except that processors P_1 and P_3 cannot fail together.

Providing such an environement model is up to the designer. His choice will depend on his knowledge of the system and the related failure assumptions. For instance, if it is unlikely for two failures to occur simultaneously, he will remove from the automaton 4(a) the three transitions from B to $F_{i,j}$. Alternatively, if he wants to consider malicious attacks, he will keep them.

3.2 Task model

3.2.1 Basic control structure pattern

Each task j is formally modeled by the LTS of Figure 5, drawn assuming that the task can be executed on the three processors of the considered architecture. It features an initial *idle* state I^j, a *ready* state 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 event t^j: this is what we mean by periodic tasks. Such periodic tasks can be directly and easily modeled by Mode Automata [15].

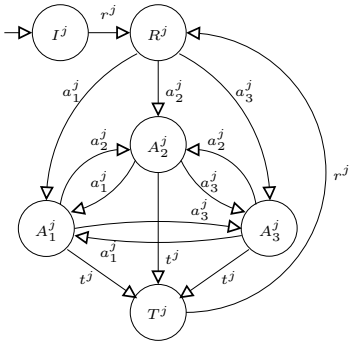

Fig. 5: Task invocation model.

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 could be decided as a reaction to a processor failure (in which case the task does not need to be stopped of course). But it could also serve to balance the load between several active processors, or to comply to the energy consumption bound of a processor.

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

3.2.2 Quantitative characteristics

Some interesting characteristics can be modeled as weights associated with states [18]; 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. Since the tasks are executed periodically, we assume that, when a task migrates from one processor to another one, its execution restarts

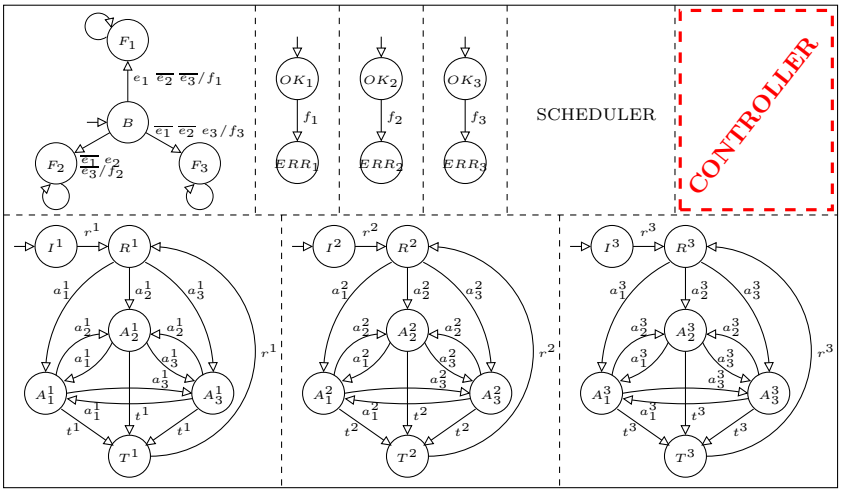

Fig. 6. A complete system with 3 processors and 3 tasks, and a controller.

from the beginning on its new processor. Hence its new processor must fully accept the task's CPU load.

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 [7]. 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.

		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.

3.3 Application model

An application is built upon the invocation of a set of tasks, considering it as a 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 [1,15]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: $Q = \sum_i \sum_j Q_i^j$.

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.

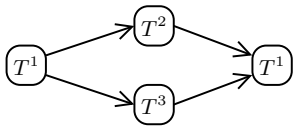

Fig. 7: Precedence constraints.

If we consider the example of Figure 7, the scheduler first issues r^1, then after receiving t^1, it issues r^2 and r^3, therefore executing T^2 and T^3 in parallel, and finally, once it has received t^2 and t^3, it issues r^1.

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 6 for a complete system made of 3 processors and 3 tasks.

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.

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 allways 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 configurations such that* $\bigwedge_i T^i$ *are reachable.*

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

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].

4.3 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 it fails, since the DCS tool explores all its state space (be it symbolically), it means that no solution exists for these failures to be tolerated and bounds on the processors' consumption.

5 Illustrative scenarii

5.1 Property 1: consistent execution

In our example, and as illustrated in Figure 8, 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 ($4 \leq 6$) (hence not taking into account the precedence constraints of Figure 7). Then, if P_2 crashes, T^2 is forced to migrate either onto P_1 or onto P_3. However, none

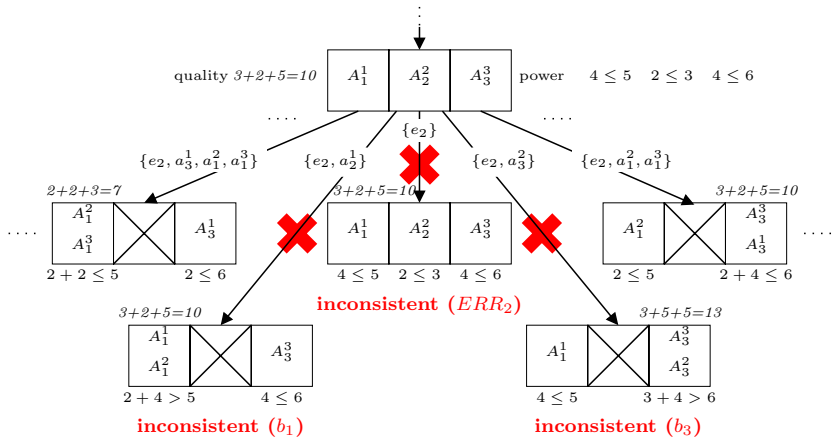

Fig. 8. Example of states (only configuration is shown) and transition control.

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 pathological 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 4-a), everything works fine, as capacity is sufficient

on any group of two remaining processors;

- for a two faults model (Fig 4-b), capacity of P_2 is insufficient to accommodate for task T^1, therefore no controller can insure functionality whatever the sequence of faults and requests;
- for the fault pattern example (Fig 4-c), a solution can be found, as the pathological processor configuration (P_2 only survivor) is not considered.

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

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 2.2, there may be several solutions with equal weights. The example in Figure 8 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.

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] [15] was used for writing the model of our systems as sets of mode automata, while the symbolic model-checker and DCS tool SIGALI [4] [17] was successfully used to automatically synthesize fault-tolerant systems from a high-level specification, and SIGALSIMU was then used to co-simulate the system and the controller, as illustrated in Figure 9.

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

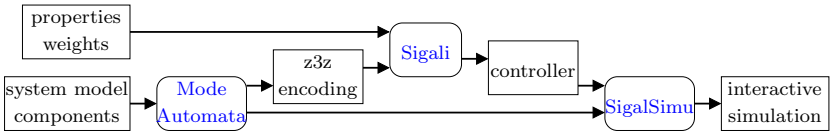

Fig. 9. Tools used.

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

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

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.

From the point of view of fault-tolerance, 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, 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.

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, ...
- control in order to optimize the cost on transitions, modeling for instance the cost of migrating the tasks, or cumulated on *paths* between significant sets of states, like start and end states,
- the same system can sometimes be reused in different environments: for

example, an image processing coder/decoder sub-system in a system-on-chip, can be embedded into different devices, e.g., a home DVD player, where power supply is not at all an issue, or a cell phone or cam-recorder, where power is indeed crucial; reusing the same model submitted to different synthesis objectives opens perspectives in further applications of these techniques, in the framework of platform-based design.

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] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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