Automating the Addition of Fault Tolerance with Discrete Controller Synthesis

Alain Girault, Éric Rutten

Automating the Addition of Fault Tolerance with Discrete Controller Synthesis

Alain Girault* and Éric Rutten**

INRIA and Grenoble University

Abstract. Discrete controller synthesis (DCS) is a formal approach, based on the same state-space exploration algorithms as model-checking. Its interest lies in the ability to obtain automatically systems satisfying by construction formal properties specified a priori. In this paper, our aim is to demonstrate the feasibility of this approach for fault tolerance. We start with a fault intolerant program, modeled as the synchronous parallel composition of finite labeled transition systems; we specify formally a fault hypothesis; we state some fault tolerance requirements; and we use DCS to obtain automatically a program, having the same behavior as the initial fault intolerant one in the absence of faults, and satisfying the fault tolerance requirements under the fault hypothesis. Our original contribution resides in the demonstration that DCS can be elegantly used to design fault tolerant systems, with guarantees on key properties of the obtained system, such as the fault tolerance level, the satisfaction of quantitative constraints, and so on. We show with numerous examples taken from case studies that our method can address different kinds of failures (crash, value, or Byzantine) affecting different kinds of hardware components (processors, communication links, actuators, or sensors). Besides, we show that our method also offers an optimality criterion very useful to synthesize fault tolerant systems compliant to the constraints of embedded systems, like power consumption.

Keywords. fault tolerant systems, discrete controller synthesis, automatic fault tolerance.
Category C.3: Computer Systems Organization. Special-purpose and application-based systems. Real-time and embedded systems.
Category D.3.2: Software. Programming languages. Concurrent, distributed, and parallel languages.
Category F.2.2: Theory of Computation. Analysis of algorithms and problem complexity. Non numerical algorithms and problems. Computations on discrete structures.

1 Introduction

1.1 Discrete controller synthesis

Discrete controller synthesis (DCS, also known as "supervisory control of discrete event systems") was invented by Ramadge and Wonham in the nineteen eighties [48]. Its theoretical foundation is language theory. The goal of DCS is, starting from two languages $\mathcal{U}$ and $\mathcal{D}$, to obtain a third language $\mathcal{C}$ such that:

$$\mathcal{U} \cap \mathcal{C} \subseteq \mathcal{D} \tag{1}$$

The three languages $\mathcal{U}$, $\mathcal{D}$, and $\mathcal{C}$ represent respectively the **plant**, the **desired system**, and the **controller**. $\mathcal{U} \cap \mathcal{C}$ is called the **controlled system**. Since the context is language theory, the alphabet $\mathcal{I}$ of the plant $\mathcal{U}$ is to be understood as the set of events that can occur, and the language $\mathcal{U}$ is the set of all possible words made with the letters of $\mathcal{I}$, each understood as a possible behavior of the plant (i.e., a sequence of events). A tool suite for DCS, called TCT, has been implemented.[1]

Since $\mathcal{U}$ and $\mathcal{D}$ are given and we want to find $\mathcal{C}$ such that $\mathcal{U} \cap \mathcal{C} \subseteq \mathcal{D}$, the solution could be written as $\mathcal{C} = \mathcal{D}.\mathcal{U}^{-1}$ provided that the operators '.' and '$^{-1}$', classical for real numbers, existed for languages. In this sense DCS can be seen as an inversion problem.

The alphabet $\mathcal{I}$ of the language $\mathcal{U}$ is partitioned into two subsets: the set $\mathcal{I}_C$ of **controllable** events and the set $\mathcal{I}_U$ of **uncontrollable** events. The first key point of DCS is that the controller can only act on the controllable

* INRIA Grenoble Rhône-Alpes and Grenoble University, POP ART project-team and LIG laboratory, 38334 Saint-Ismier cedex, France, Email: Alain.Girault@inria.fr. This research was supported by a Marie Curie International Outgoing Fellowship within the 7th European Community Framework Programme.
** INRIA Grenoble Rhône-Alpes and Grenoble University, SARDES project-team and LIG laboratory, 655 avenue de l'Europe, 38334 Saint-Ismier cedex, France, Email: Eric.Rutten@inria.fr
[1] TCT: http://www.control.utoronto.ca/people/profs/wonham.

events of the plant. The second key point is that the synthesized controller is **the most permissive one**, meaning that the language $\mathcal{U} \cap \mathcal{C}$ must be the greatest one included in $\mathcal{D}$.

Note that DCS can fail for a given objective $\mathcal{D}$. This means that no language $\mathcal{C}$ exists acting only on $\mathcal{I}_C$ and such that $\mathcal{U} \cap \mathcal{C} \subseteq \mathcal{D}$.

Finally, it is classical to represent the pair (plant,controller) as a **closed loop system**, where the controller observes the plant and modifies its behavior through the controllable events. This is illustrated by Figure 1. This figure depicts two arrows between the plant $\mathcal{U}$ and the controller $\mathcal{C}$. These communications do not necessarily need to take place through a communication network since most DCS tools directly produce the controlled system $\mathcal{U} \cap \mathcal{C}$.

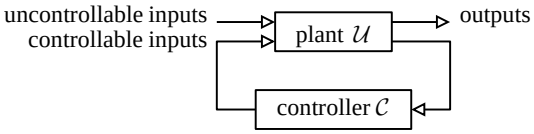

Fig. 1. The closed loop system (plant,controller).

Several research teams have applied and extended these language theory techniques to **labeled transition systems** (LTS), both in the area of computer science and discrete event systems control theory. The algorithms used in DCS are the same as those of model checking: mostly it is state space exploration, reachability analysis, and invariance analysis, be it enumerative or symbolic with Binary Decision Diagrams (BDDs). In particular, this is the case of the SIGALI [2] [43] tool that we have used in the present article. Also, in model checking, it is well known that objectives can be equivalently expressed as predicates on the states of $\mathcal{U}$ or as LTSs.

Within SIGALI, the desired system $\mathcal{D}$ is specified as a set of state properties, possibly involving synchronous observers [29], and synthesis objectives upon them : we use essentially the objectives to **make invariant** a subset of states, or **keep reachable** a subset of states. This is very versatile and allows the user to change easily his synthesis objective. Figure 2 summarizes the behavior of SIGALI.

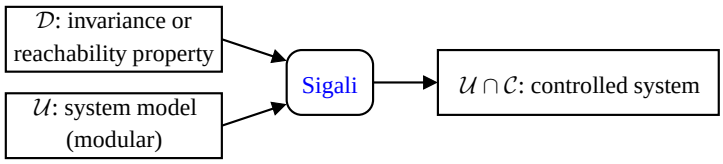

Fig. 2. Overview of DSC with SIGALI.

DCS can be used on different kinds of systems, be it hardware or software. In the case of a software system, modeling the system with an LTS does require a high level of abstraction. This is classically achieved by considering only the control layer of the software and by abstracting away the data computations, as in [1].

1.2 The need for fault tolerance

There is no arguing that dependability is a key issue in critical systems. There are three threats to dependability: fault, error, and failure, with the classical causality relationship [4]:

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

For instance, consider a software where one variable `x` is incorrectly modified in one execution path. This is commonly known as a bug, but, in the field of dependability, it is referred to as a fault. When the software takes this precise execution path, then it is an error. When the incorrect value of `x` prevents the software from delivering its nominal service, then we have a failure. Finally, the failure of a subsystem is seen as a fault in the encompassing system.

We believe in the need of separation of concerns between the functional specification and the fault tolerance requirement. Hence, we would like to propose *automatic* methods to turn a fault intolerant program implementing the functional specification[3], into a new program implementing the same functional specification (i.e., preserving the semantics of the initial program) and tolerant to the faults required by the user. There have been several methods proposed in the past, and we will study them in Section 8.

[2] SIGALI: `http://www.irisa.fr/vertecs/Logiciels/sigali.html`.

[3] By "fault intolerant", we mean a program that is not necessarily fault tolerant.

1.3 Contribution

We propose a DCS-based framework to transform automatically a fault intolerant program into a fault tolerant one. It offers the following features:

- The possibility to try several fault hypotheses on the same specification.
- The possibility to evaluate several fault tolerance requirements.
- In the final program, the guarantee by construction of the fault tolerance level required by the user.

The above-mentioned features are generally also offered by most methods and algorithms that provide fault tolerance automatically. The originality of our DCS-based method is that the failure recovery mechanism provided by DCS is *dynamic* (hence it does not induce too much redundancy overhead like static methods), with a *static* guarantee on the fault tolerance of the obtained system (unlike dynamic methods). As a result, if offers the best of both worlds, static guarantee and small overhead, at the price of an exhaustive state-space exploration at compile-time. Besides, our method also offers an optimality criterion very useful to synthesize fault tolerant systems with embedded constraints like power consumption.

Compared to relevant related work on approaches similar to DCS for fault tolerance, our originality is that we propose an integrated framework offering a full coverage of the possible failures of the system's components: processors, communication links, actuators, and sensors. Furthermore, we not only address the easy to tolerate crash failures, but also the much more difficult value and Byzantine failures, a feature which is unique to our framework. Finally, we use optimal DCS over finite paths in order to provide more possibilities of synthesis for fault tolerance; this is another unique feature of our framework.

1.4 Outline

We start by introducing the formal model of labeled transition systems and how they are used in DCS: this is Section 2. Then, we present in Section 3 the general principles for automating the addition of fault tolerance with DCS. In Section 4 we detail how to specify and handle the failures of hardware components (processors, communication links, actuators, and sensors). In Section 5 we detail how to specify and handle several kinds of failures (crash, value, and Byzantine failures). In Section 6 we present advanced DCS features, like how to specify and handle quantitative constraints, and how to obtain a distributed controller. Then, we present in Section 7 two previously unpublished case studies that exemplify how our framework can be actually used to specify and make fault tolerant an entire system. We have completed three other case studies that further demonstrate the pertinence of our framework: they have been published in [27,23,28,22] so we do not include them in the present article (although several of our examples presented in Sections 4 to 6 will be taken from these articles). We end with a presentation of the related work in Section 8, and with concluding remarks in Section 9.

2 Formal models used in DCS

2.1 Labeled transition systems

A **labeled transition system** (LTS) is a tuple $S = \langle \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} \rangle$, where $\mathcal{Q}$ is a finite set of states, q_0 is the initial state of S, $\mathcal{I}$ is a finite set of input events (produced by the environment), $\mathcal{O}$ is a finite set of output event (emitted towards the environment), and $\mathcal{T}$ is the transition relation, that is a subset of $\mathcal{Q} \times \mathcal{B}ool(\mathcal{I}) \times \mathcal{O}^* \times \mathcal{Q}$, where $\mathcal{B}ool(\mathcal{I})$ is the set of boolean expressions of $\mathcal{I}$. If we denote by $\mathbb{B}$ the set $\{true, false\}$, then a guard $g \in \mathcal{B}ool(\mathcal{I})$ can be equivalently seen as a function from $2^{\mathcal{I}}$ into $\mathbb{B}$.[4]

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), and where $a \in \mathcal{O}^*$ is a conjunction of outputs that are emitted when the transition is taken (a is the **action** of the transition). State q is the **source** of the transition (q, g, a, q'), and state q' is the **destination**. A transition (q, g, a, q') will be graphically represented by $q \xrightarrow{g/a} q'$.

An LTS is **deterministic** (resp. **reactive**) iff, for each state $q \in \mathcal{Q}$ and for each valuation of the inputs, there exists at most (resp. at least) one transition from q and whose guard is true for this inputs valuation.

[4] For any set X, 2^X is the set of all subsets of X.

The composition operator of two LTSs put in parallel is the **synchronous product**, noted $||$, as defined by Milner [46] and a characteristic feature of the synchronous languages [6]. The synchronous product is commutative and associative. Formally:

$$\langle \mathcal{Q}_1, q_{0,1}, \mathcal{I}_1, \mathcal{O}_1, \mathcal{T}_1 \rangle \,||\, \langle \mathcal{Q}_2, q_{0,2}, \mathcal{I}_2, \mathcal{O}_2, \mathcal{T}_2 \rangle = \langle \mathcal{Q}_1 \times \mathcal{Q}_2, (q_{0,1}, q_{0,2}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T} \rangle$$
$$\text{with } \mathcal{T} = \{((q_1, q_2) \xrightarrow{g_1 \wedge g_2 / a_1 \wedge a_2} (q_1', q_2')) \mid q_1 \xrightarrow{g_1/a_1} q_1' \in \mathcal{T}_1, q_2 \xrightarrow{g_2/a_2} q_2' \in \mathcal{T}_2\}.$$

Here, (q_1, q_2) is called a **macro state**, where q_1 and q_2 are its two **component states**.

Like all product operators for LTSs, the synchronous product causes a combinatorial explosion, since the number of states in $S_1||S_2$ is, at worst, equal to the product of the number of states of S_1 by S_2. However, it limits this explosion, compared to the asynchronous product, where for two LTSs making each a transition in parallel, all interleavings are explicitly represented in the product, with all intermediary states. Indeed, the synchronous product makes it possible to group parallel transitions into one global transitions where several local transitions are taken in the same step, without developing sub-steps.

A **path** in the LTS $S = \langle \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} \rangle$ is a sequence of transitions $q_1 \xrightarrow{g_1/a_1} q_2 \xrightarrow{g_2/a_2} q_3 \cdots q_n \xrightarrow{g_n/a_n} q_{n+1}$. A **trace** is a path starting in the initial state q_0. The set of all traces of S is noted $T(S)$. A state q of $\mathcal{Q}$ is **reachable** iff there exists a trace to q. A set of states E is **reachable** iff all its states are. In the CTL temporal logic [24], this is stated as $S \vdash \forall\Diamond(E)$. A set of states E is **invariant** iff any transition having as source a state of E has its destination state in E. In CTL, this is stated as $S \vdash \forall\Box(E)$.

2.2 Discrete controller synthesis on labeled transition systems

The plant $\mathcal{U}$ is specified as an LTS, more precisely the result of the synchronous product of several LTSs. $\mathcal{D}$ is the objective that the controlled system must fulfill. The controller $\mathcal{C}$ obtained with DCS achieves this objective by restraining the transitions of $\mathcal{U}$, that is, by disabling those that would jeopardize the objective $\mathcal{D}$.

The set $\mathcal{I}$ of inputs of $\mathcal{U}$ is partitioned into two subsets: the set $\mathcal{I}_C$ of controllable inputs and the set $\mathcal{I}_U$ of uncontrollable inputs. Formally, $\mathcal{I} = \mathcal{I}_C \cup \mathcal{I}_U$ and $\mathcal{I}_C \cap \mathcal{I}_U = \emptyset$. As a consequence, a transition guard $g \in \mathcal{B}ool(\mathcal{I}_C \cup \mathcal{I}_U)$ can be seen as a function from $2^{\mathcal{I}_C} \times 2^{\mathcal{I}_U}$ into $\mathbb{B}$.

A transition is **controllable** iff there exists at least one valuation of the controllable inputs such that its guard is false; otherwise it is **uncontrollable**. Formally, a transition $(q, g, a, q') \in \mathcal{T}$ is controllable iff $\exists X \in 2^{\mathcal{I}_C}$ such that $\forall Y \in 2^{\mathcal{I}_U}$, we have $g(X, Y) = false$.

In the framework of this paper, we use the following functions to synthesize the controlled system $\mathcal{U} \cap \mathcal{C}$, where E is any subset of states of $\mathcal{U}$ (possibly specified itself as a predicate on states φ):

- $S' = make\_invariant\,(S, E)$ is a function that synthesizes and returns a controllable system S' such that the controllable transitions leading to states $q_{i+1} \notin E$ are inhibited, *as well as* those leading to states from where a sequence of uncontrollable transitions can lead to such states $q_{i+k} \notin E$.

  For example, consider a LTS S, synchronous composition of several LTSs, with one of them being an observer with a state $Error$. The function making invariant the set of global states where the local state of the observer is different from $Error$ inhibits behaviors leading to this $Error$ state, making it unreachable. This technique will be used in Section 6.4.
- $S' = keep\_reachable\,(S, E)$ is a function that synthesizes and returns a controlled system S' such that the controllable transitions entering subsets of states from where E is not reachable are disabled. Note that making E invariant is equivalent to making states not in E unreachable.

  For example, a system can have a set of states defined as safe back-up configurations, where the system should always be able to go in case of need, from anywhere in its reachable state space.
- $E' = reachable\_under\_control\,(S, E)$ is a function that returns a subset E' of the states of S such that states in E' are reachable by controllable transitions. This function allows us to transform a reachability objective into an invariance one: $keep\_reachable\,(S, E) = make\_invariant\,(S, reachable\_under\_control\,(S, E))$. This feature will be useful in Section 6.3 when we consider conditioned reachability objectives.

It must be noted that the order in which synthesis operations are applied does matter: indeed, their composition is *not commutative* in general. 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.

This introduction to the formal models used in DCS is kept simple in order to concentrate the paper on its contribution concerning fault tolerance issues. Readers interested in more detailed formalizations of discrete controller synthesis are referred to [48,43,2].

Figures 3 shows, from left to right, an example of DCS on an LTS with five states and two inputs, one controllable c and one uncontrollable u. The objective is to make this system invariant w.r.t. the subset of state E, i.e., to avoid state S4. Given the particular uncontrollable transitions, only a smaller subset E′ can be controlled. This example shows that, in the general case, even for propositional state properties and invariance objectives, the DCS algorithm has to explore the whole state space in order to find the controllable transition in a path where control has to be enforced, in cases where the following transitions are not controllable. The LTS on the right shows the controlled system, where in state S2 the controllable input c is forbidden to take the value $true$ and must be $false$, hence inhibiting the wrong behavior to be avoided by disabling the transition from S2 to S3.

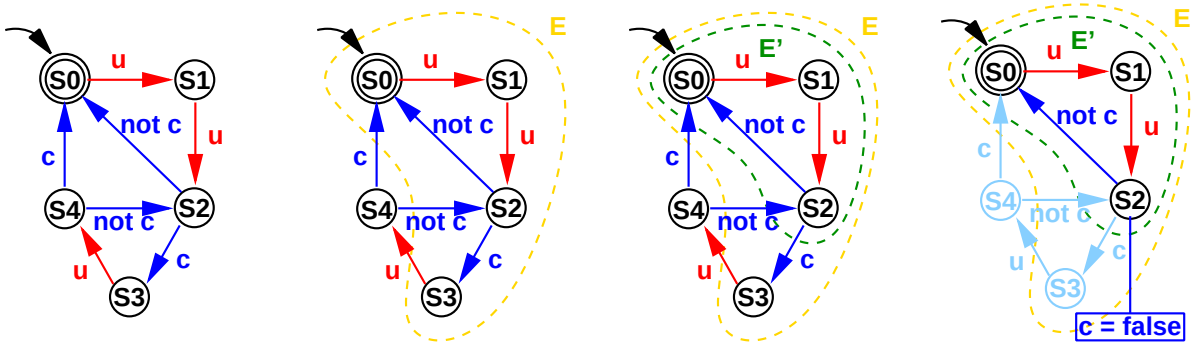

Fig. 3. An example of DCS, on an automaton.

If DCS fails w.r.t. the objective $\mathcal{D}$, since all the state space is traversed during the synthesis (be it exhaustively or symbolically), it means that it is impossible to restrain the plant $\mathcal{U}$ only by disabling controllable transitions. In our framework for fault tolerance, we will discuss the implication of this situation.

2.3 Tools and programming languages

We use the Mode Automata language to program LTSs [40].[5] Without going into too many details, Mode Automata are LTSs: each state represents a different mode of operation of the program, specified as data-flow equations relating the inputs and the outputs of the program. Mode Automata use the synchronous product operator to combine several programs put in parallel. This allows the user to program in a clean and modular way.

The compiler associated to the Mode Automata language, MATOU, compiles an LTS into the $\mathbb{Z}/3\mathbb{Z}$ format[6], which is the input format of the SIGALI tool for DCS [43]. Finally, we use SIGALSIMU to co-simulate the system and the controller. This tool chain, illustrated in Figure 4, is the support for a DCS methodology [1] that was also used to generate task managers [44].

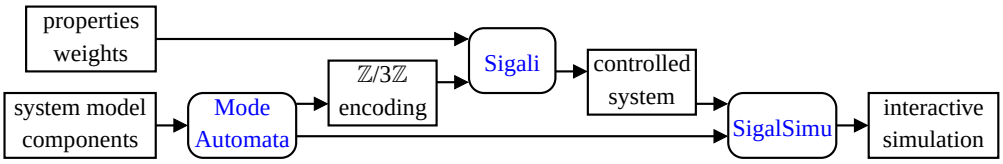

Fig. 4. Tools used.

Note that we use the synchronous composition as a model specification facility, because of its clarity and elegance; however it is not intrinsically necessary, and the asynchronous product may be used too; in that sense our choice of tools is not central to the paper. Concerning performance, the impact is not obvious: synchronous composition tends to reduce the state space because intermediate states in the communications are abstracted away by the instantaneous broadcasting; on the other side, labels on the transitions can be more complex.

[5] Matou: http://www-verimag.imag.fr/~maraninx/MATOU

[6] $\mathbb{Z}/3\mathbb{Z}$ is the Galois field with three elements, $\{-1, 0, 1\}$.

3 A framework for automating the addition of fault tolerance with DCS

From the point of view of fault tolerance, it is natural to consider the fault events as uncontrollable events. Then, the plant $\mathcal{U}$ must represent all the possible behaviors, both the good ones (where either no fault occurs, or those that occur are masked) and the bad ones (where at least one fault prevents the system from providing its nominal service). Finally, the desired system $\mathcal{D}$ must express the fact that a certain number of faults must be tolerated. By synthesizing a controller $\mathcal{C}$ guaranteeing that $\mathcal{U} \cap \mathcal{C}$ satisfies the properties of $\mathcal{D}$, we will obtain *automatically* a fault tolerant system.

Note however that uncontrollable events are by no means restricted to be fault events. They can be any event that the user wants to be determined by the environment, e.g., non-deterministic events.

Besides, the fault model will be described as an LTS that will be composed in parallel with the remaining of the plant specification. This approach yields two advantages, first it is flexible and modular since it is possible to change the fault hypothesis without modifying the remaining of the specification, and second it is formal thanks to the usage of an LTS.

The design of dependable systems calls for a dedicated specification and validation procedure. Two key points must be taken into account: the fault hypothesis and the fault tolerance policy. This is detailed in the following paragraphs.

3.1 Defining the fault intolerant system

The first step involves designing the fault intolerant system. We use LTSs to specify the various concurrent parts of the system (both hardware and software), and the synchronous parallel composition operator to compose them in order to obtain the full system; these formal models have been defined in Section 2.1. We advocate that designing a single monolithic LTS is both non-modular and non-scalable. Breaking down the system into a set of concurrent components that collaborate together to the desired behavior has always been the method of choice to achieve modularity and scalability. It is also much easier to design each sub-component independently of its interactions with the other sub-components, and to rely on DCS to derive automatically their interactions.

3.2 Defining the fault hypothesis

A fault hypothesis states which components of the system may fail. If more than one component is likely to fail, **failure configurations** are a common way to express subsets of components that may fail together. According to this hypothesis, the remaining components are supposed to be reliable: they never fail, or if such a failure occurs, the whole system fails. The fault hypothesis can be obtained by a stochastic analysis, in order to find the probability for each failure configuration; this is out of the scope of our paper. In the following, we assume that all the specified failure configurations are equally probable.

Then, a fault model is required for each component identified by the fault hypothesis. For a given component failure, what this failure implies has to be specified. This amounts to defining a behavior that is triggered by this failure. For instance, when a component fails (processor, communication link, sensor, etc.), it may stop reacting to its environment (fail-silent behavior) or it may react by emitting random values (Byzantine behavior). Another aspect of the fault model is to specify whether the faults are permanent or temporary. The failure models must be combined with the failure patterns in order to specify realistic failure scenarios.

3.3 Defining a fault tolerance policy

Ideally, a fault tolerant system should maintain its functionalities and its performance (nominal service) even though some of its components are faulty. In practice, this assumption is too strict and expensive to implement. Thus, if a failure occurs, the nominal service may be replaced by a **degraded operating mode**. When the system runs inside a degraded mode, only a subset of its initial functional requirements are still met. We achieve fault tolerance by establishing such a degraded mode when a failure occurs.

In our context, DCS is used to control the system's behavior, in order to ensure a minimal service. The fault tolerance policy is a DCS **control objective** expressing fault tolerance: what the system should always do or avoid, despite failure occurrences. The control objective can be a temporal logic property, expressing either an invariant or an accessibility property.

However, a degraded operating mode only exists for those systems that are **controllable**. DCS will act on the subset of **controllable events** of the system for service maintenance purposes. The system behavior will be

constrained by driving these controllable events appropriately. If DCS fails, it means that the system at hand cannot be made fault tolerant for the required fault tolerant policy and under the specified fault hypothesis; thus, the system must be redesigned, either by relaxing some constraint or by adding/improving the available resources: for instance, the number of processors can be increased.

One important point not addressed by DCS is the possible faults of the controller itself. One feasible solution is to apply to the controlled system the classical techniques of fault tolerance, for instance active replication with voting, where the number of active replica depends on the number of faults to be tolerated.

Technically, the fault tolerance policy is specified in terms of the functions $make\_invariant\,(S, E)$ and $keep\_reachable\,(S, E)$, where E is any subset of states of the fault intolerant system S. This subset E will be specified either directly as a set or as a predicate φ on states of S. In particular, when the fault intolerant system S is the parallel product of several LTSs, then φ can be a predicate on the states of one (or several) of its component LTSs:

$$\mathcal{U} = S_1 \parallel S_2 \parallel \cdots \parallel S_n \quad \text{with} \quad S_i = \langle \mathcal{Q}_i, q_{i0}, \mathcal{I}_i, \mathcal{O}_i, \mathcal{T}_i \rangle \tag{2}$$

$$E = \{q = (q_1, \ldots, q_n) \in \mathcal{Q}_1 \times \cdots \times \mathcal{Q}_n \mid \varphi(q_1, \ldots, q_n) = true\} \tag{3}$$

In other words, each macro state of the product such that its component states match the predicate φ is in E.

4 Specifying the hardware component failures

A complete system classically consists of several hardware components: processors, communication media, sensors, and actuators. In accordance, when designing fault tolerant systems, we should not only address the failures of the processors (what most of the related work does), but also the failures of the communication media, the sensors, and the actuators. In particular, in distributed systems, communication media are usually more subjects to failures than processors. Also, in embedded systems, sensors and actuators are critical components, whose failure will inevitably put the system in a degraded mode, if not causing its actual failure.

As we have said in Section 3, which hardware components can fail will be expressed in the fault tolerant hypothesis. We show in this section that each kind of hardware component calls for specific means to handle its failure.

4.1 Processor failures

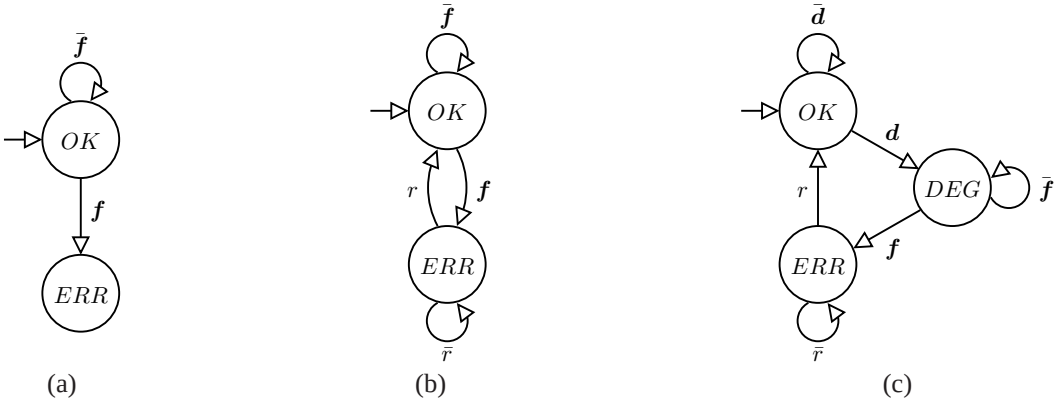

Fig. 5. (a) LTS of a processor with permanent fail-silent failures; (b) Same with temporary failures; (c) Same with degraded modes.

Figure 5(a) specifies a processor subject to permanent failures: the processor starts in the OK states, and upon the reception of the input failure event $\boldsymbol{f}$, goes into the ERR state, where it stays forever. Figure 5(b) specifies a processor subject to transient failures: once in the ERR state, it can go back to the OK state following the repair input event r. Figure 5(c) specifies a processor that can go into the DEG state following the degraded mode input event $\boldsymbol{d}$; once there, it can go into the ERR state; finally, the processor can be repaired (event r). Degraded modes are very useful to model intermediary behaviors where the processors is not crashed but does not deliver its full

functionality: for instance, it could be running at half its normal clock speed. These three LTSs are just examples of what can be specified, and the user is free to modify them to suit his needs.

In terms of DCS, it is natural that the events $\boldsymbol{f}$ and $\boldsymbol{d}$ be uncontrollable (i.e., $\in \mathcal{I}_u$), since a failure is an event intrinsically uncontrollable. To differentiate them from the other events, they are typeset in bold italic font. Concerning the repair event r, this depends on the system the user wants to specify: if the system is self-repairable, then r will be controllable, while if the repair is an external operation (e.g., requiring the intervention of a human operator), then it will be uncontrollable.

If we are dealing with a distributed architecture consisting of n processors, then we must put in our specification n such LTSs, not necessary all of the same kind (the three LTSs above can be mixed at will). Each such LTS will need to have a separate vocabulary, each identified by a different subscript: $\boldsymbol{f_i}$ will therefore denote the failure event of processor i.

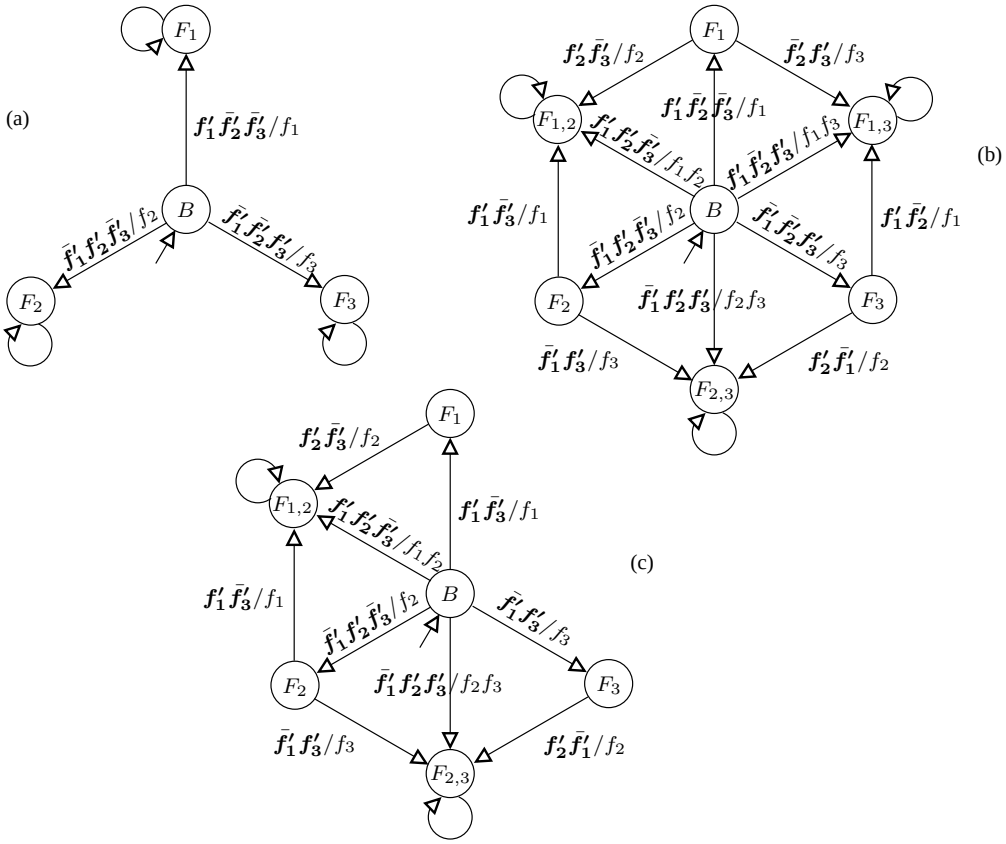

Fig. 6. Three examples of environment models for a 3 processor architecture: (a) Only one failure can occur; (b) Two failures can occur, possibly simultaneously; (c) Failure pattern.

Aside from the processor failure model, what failures can occur in the system must also be specified: for instance, how many processors can fail? Or can they fail simultaneously? In terms of our processor LTSs of Figure 5, the question is how can the f_i and d_i events occur? Like we have said, all the failure events f_i and d_i are uncontrollable. But this means that there is no constraints whatsoever on them. In particular, *all* the 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. To specify the way in which the failures can occur, the user must provide a LTS modeling the environment. Its purpose is to issue the signals f_i (resp. d_i) from signals $\boldsymbol{f_i'}$ (resp. $\boldsymbol{d_i'}$) produced by the environment. These signals $\boldsymbol{f_i'}$ and $\boldsymbol{d_i'}$ 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 and d_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.

The three LTSs of Figure 6 concern a distributed architecture consisting of three processors: they are examples of possible environment models that filter the uncontrollable events $\boldsymbol{f_i'}$ and $\boldsymbol{d_i'}$ to produce the local events f_i and d_i that must be tolerated by the system. Providing such an environment 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 6(b) the three transitions from B to $F_{i,j}$. Alternatively, if he wants to consider malicious attacks, he will keep them.

The models above alone do not allow the user to specify how the failures are actually detected. If the user wants to concentrate on error processing only, then this is sufficient and he can assume that there exists a reliable external unit that reports an error if and only if a failure has occurred. Otherwise, the user can specify additional LTSs to model the failure detection process, for instance by detecting discrepancies between the outputs of two redundant processing units, and issuing the corresponding $\boldsymbol{f}'_i$ event.

To specify a whole system, the above-described models of processors and environments can be used along with the models of several tasks running on those processors. Such a task can, for instance, be migrated from a processor P1 to another processor P2 in order to react to a failure event affecting P1. The advantage lies in the decoupling of the task, the environment, and the processor model, making the definition of the fault tolerant policy straightforward: indeed, it suffices to request that no task be active on a faulty processor to synthesize automatically, by DCS, a controller making the system fault tolerant. Such a scheme has been reported in [27]. Additionally, more sophisticated fault tolerance mechanisms can be specified, for instance checkpointing and rollback, as described in [22].

Finally, note that communication links and memories can be treated exactly in the same way.

4.2 Actuator failures

In order to model the failure of an actuator, one has to specify how the failures affect the service that the actuator is supposed to deliver. For instance, consider a braking system subject to failures. When not faulty, the brake is either open (state O) or closed (state C), and can switch from the open to the close state according to the controllable input c, or vive versa with o. The brake becomes faulty following the uncontrollable event $\boldsymbol{f}$, and goes either in the FO state if it was open at the time of the failure, or to the FC state if it was closed. The failures are permanent, which is modeled by the fact that the states FO and FC are sink states. This behavior is encoded in the LTS of Figure 7(a). To model temporary failures, it suffices to add transitions from the FO (resp. FC) state back to the O (resp. C) state, labeled with the controllable repair input r. This is depicted in Figure 7(b).

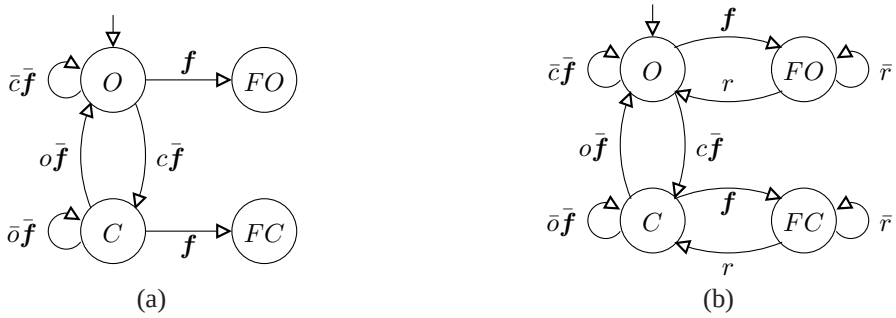

Fig. 7. (a) LTS of a braking system subject to permanent failures; (b) Same with temporary failures.

Furthermore, degraded modes can also be specified: for instance, the LTS of Figure 8(a) shows a braking system that can be either open (state O), closed (state C), or half-open (state H). In terms of breaking pressure, the pressure would of course be equal to the maximal pressure (say m) in state F, equal to 0 in state O, and equal to $m/2$ in state H. Figure 8(b) shows the LTS of a braking system with two degrade modes, one consisting of the states DO and DH, where the breaking pressure is in the interval $[0, m/2]$, and a second degraded mode consisting of the states DC and DH', where the braking pressure is in the interval $[m/2, m]$. The degraded states are entered upon the occurrence of the uncontrollable event $\boldsymbol{d}$, while the failure states are entered upon the occurrence of the uncontrollable event $\boldsymbol{f}$. For the sake of clarity, the self-loops have been omitted.

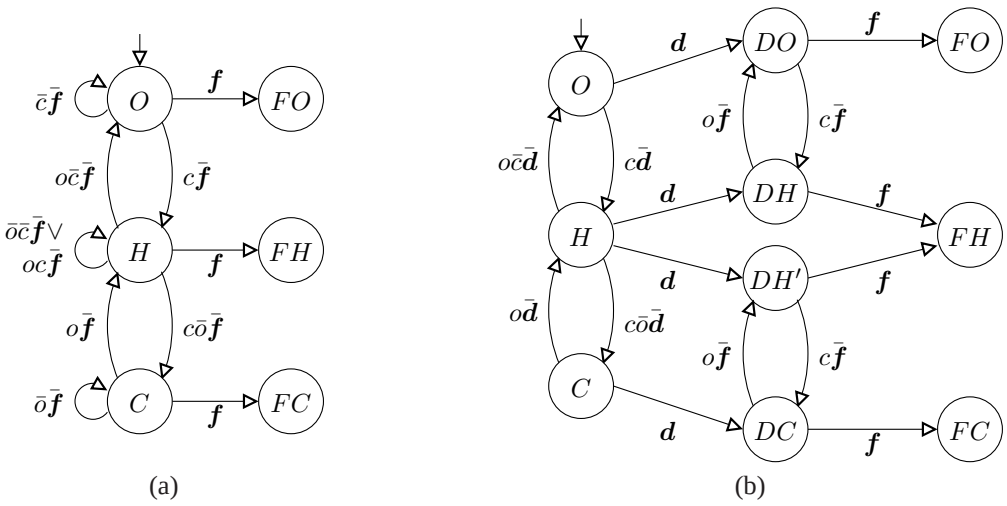

Fig. 8. (a) LTS of a three-state braking system subject to permanent failures; (b) Same with degraded modes.

Like in Section 4.1, it is mandatory to specify how many actuators of the same kind can fail and in what manner. This is done by providing an LTS like those of Figure 6.

Other kinds of actuators can be defined. The common feature is that the LTS must specify how the failures affect the service that the actuator is supposed to deliver. For instance, a valve controlling the flow of some liquid might be specified by exactly the same LTS as the braking system that we have just described.

The common feature that all actuator specifications must share concerns the state variables of the actuator (e.g., the braking pressure, or the flow of liquid that passes through the valve). Due to the DCS framework, these state variables must be encoded by *discrete* variables. Yet, as we have shown in Figure 8(a), it is perfectly possible to extend this discretization to more than two states, at the price of more state space. This is the same as the output of a sensor, as we will see in Section 4.3.

4.3 Sensor failures

In order to model a sensor subject to failures, it is necessary to specify how the failures affect the service that the actuator is supposed to deliver. For instance, consider a liquid level sensor: either it is immersed in the liquid (hence wet, in state W), or it is not (hence dry, in state D), or faulty (in the ERR state). See the LTS in Figure 9(a). It goes to the ERR state upon receiving the failure event $\boldsymbol{f}$. It goes from state W to D upon receiving the event d, and back to state W upon receiving the event w. The events d and w are issued by the environment (or possibly by another LTS modeling the liquid tank itself) to signal the sensor that it must change state. Concerning the failure event $\boldsymbol{f}$, either it is uncontrollable or it must be produced by an environment model provided by the user, just like the LTSs of Figure 6 for the processor failures.

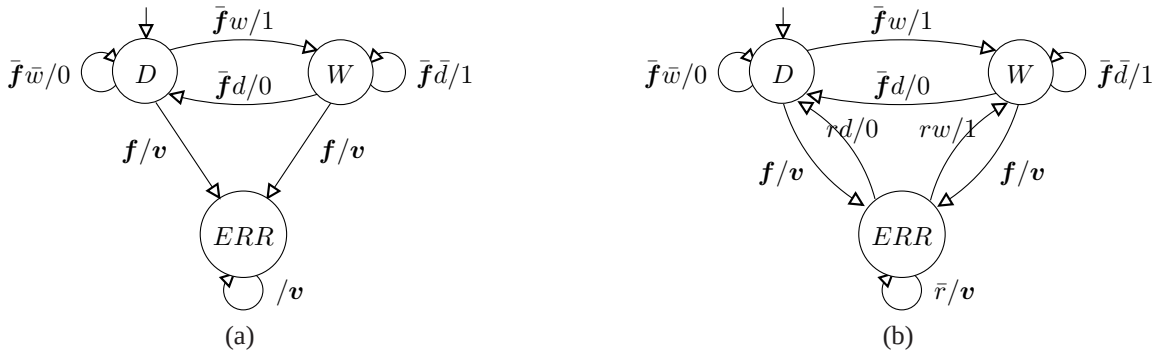

Fig. 9. (a) LTS modeling a liquid level sensor subject to permanent failures; (b) Same with transient failures.

Now, the purpose of a sensor is to produce an output corresponding to the physical data sensed by it from the environment. In the case of a liquid level sensor, this will be a Boolean equal to 1 when the sensor is wet, and to 0 when the sensor is dry. However, when the sensor is faulty, its the value of its output should not be fixed: we model this by making this output equal to an uncontrollable event, called $\boldsymbol{v}$ in Figure 9(a).

Figure 9(a) specifies a sensor subject to permanent failures: the ERR state is a sink state. If one wants to specify a sensor subject to transient failures, it suffices to add transitions back from ERR to D and W, like it is done in Figure 9(b).

5 Handling different kinds of failures

There are various kinds of failures that can affect the hardware components of the system. They are classified according to the following criteria [4]:

- **their domain**: in value or temporal (in the latter case, their duration must also be specified);
- **their coherence** w.r.t. all the users;
- **their detectability** by the user.

For instance, crash failures are actually temporal and permanent failures; they are detectable and coherent: they are the easiest failures to detect (and to tolerate), but conversely they have the least failure mode coverage[7]. At the other end of the spectrum, Byzantine failures are incoherent value failures: they are the hardest failures to detect (and to tolerate), but conversely they have the largest failure mode coverage. Because they are easier to model and to tolerate, most of the related work concentrates on crash failures. In contrast, we show in this section not only how to handle crash failures within our DCS framework, but also value failures and even Byzantine failures.

5.1 Crash failures

Crash failures are the easiest kind of failures to model and to handle. A hardware component subject only to crash failures is called **fail-silent**. Either it works fine, or it is faulty and in this case it ceases to emit any output. In particular, such failures are very easy to detect, for instance with heartbeat: the processor emits an "I am alive" message at regular intervals with some fixed period T, and whenever two messages in a row are not received, we know that the processor is faulty.

A consequence of this definition is that Figure 5 does not capture the nature of crash failures, because it says nothing about the outputs of a processor when it is faulty. But in fact, it is not necessary to model explicitly the processor's outputs: it suffices to specify that any task executed onto a processor runs fine (i.e., produces correct outputs) until the processor becomes faulty, in which case the task stops producing any outputs. In conjunction with the LTSs of Figure 5, it suffices to specify that no task can be active on a faulty processor.

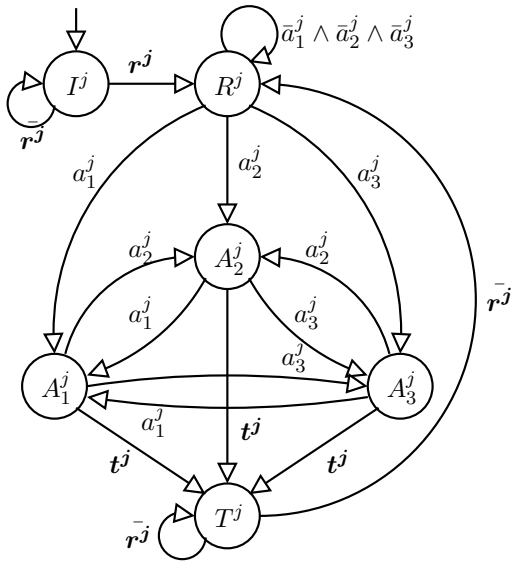

Fig. 10. LTS of task τ_j.

[7] The coverage of a failure mode assumption is defined as the probability that the assertion that formalizes the assumption is true, conditioned on the fact that the component has failed [47].

For instance, Figure 10 shows the LTS of a simple task τ^j that can be active on either processor P_1, P_2, or P_3, borrowed from [27]. The task is first idle (state I^j) until it receives a run uncontrollable event $\boldsymbol{r^j}$. It then goes to the ready state R^j, where it waits until the controller decides to activate it either on processor P_1 (state A_1^j), on processor P_2 (state A_2^j), or on processor P_3 (state A_3^j). At any time, the controller can decide to migrate the task onto another processor (thanks to the events a_i^j). This goes on until the task terminates and goes to state T^j, which is signaled by the reception of the uncontrollable event $\boldsymbol{t^j}$.

Now, when several tasks obeying to the above specification are run concurrently on a three processor architecture, if the user wants to model fail-silent failures, it suffices to express that no task should ever be active on a faulty processor, and hence should be migrated by the controller onto another not faulty processor. Since the failures of a fail-silent processor are easy to detect, it is consistent to apply DCS with the following function:

$$S' = make\_invariant\left(S, \neg \bigvee_{j=1}^{n} \bigvee_{i=1}^{p} (A_i^j \wedge ERR_i)\right)$$

where S is the fault-intolerant system resulting from putting in parallel one LTS like the one of Figure 10 for each task τ^j, one LTS like the one of Figure 5(a) for each processor P_i, and one LTS like the one of Figure 6(a) specifying the environment. This scheme has been used in [27,22].

One can notice that, in the resulting controlled system, this control objective will lead to choosing between states A_i^j in the LTS of Figure 10. Indeed, in the global LTS resulting from the composition of LTSs shown previously, from a global state product of the local state OK of a processor P_1 and the local state A_1^j of task τ^j, on the occurrence of a fault event f_1, the controller will only allow a transition satisfying the objective, and given that a local transition towards ERR_1 will take place, the controller will constrain the values of the a_i^j controllable inputs in such a way that the LTS *must* go from the local state A_1^j into either A_2^j or A_3^j. This is also illustrated in Figure 3.

5.2 Value failures

Value failures are much more difficult to detect than crash failures: in particular, obvious schemes like heart-beating do not work. The difficulty within a DCS framework is to model the fact that when the failure occurs, the variable concerned by this failure can take any value. Two cases must be distinguished:

- For a **boolean variable** u, this can be easily modeled by adding an additional uncontrollable variable $\boldsymbol{v}$ and making u equal to $\boldsymbol{v}$ whenever the failure occurs. As a result, the value of the faulty variable u can be any value in $\mathbb{B}$. This is shown in Figure 9, where the additional uncontrollable variable $\boldsymbol{v}$ serves as the output of the sensor whenever it is faulty.
- For a **numerical variable** u, it is necessary to discretize its domain of values, to encode this domain with boolean variables, and to add as many uncontrollable variables to generate uncontrollable values in the domain of u. This is exactly similar to abstract interpretation [20], and future work could concern coupling abstract interpretation with DCS in order to be able to synthesize controllers on systems with numerical variables; in particular, we plan to apply techniques such as dynamic partitioning [30].

5.3 Byzantine failures

Byzantine failures are like value failures, except that they are also incoherent [37]. This means that a processor subject to Byzantine failures which must send a data to two distinct processors can send two different values to each of them! For this reason, they are even more difficult to detect than value failures. The scheme we propose is a generalization of the value failures: for each boolean variable u computed by a given component C, we add as many additional uncontrollable variables $(\boldsymbol{v_i})_{1\leq i\leq n}$, where n is the number of other components $(C_i)_{1\leq i\leq n}$ to which C must send the value of u. When C is not faulty, it sends to all components $(C_i)_{1\leq i\leq n}$ the same correct value u. But when C is faulty, it sends to each component C_i a different value $\boldsymbol{v_i}$.

Figure 11(a) depicts a non-faulty component C that computes an internal function $u = F(i)$ and transmits the result u to n other components $C_1, \ldots, C_n$: each of those components receives the same value. Figure 11(b) shows the corresponding component having the same functionality F but with Byzantine failures. When the failure event $\boldsymbol{f}$ occurs (i.e., when $\boldsymbol{f} = 1$), the result u of the internal function F is bypassed, and each component C_i receives a different value $\boldsymbol{v_i}$ instead of u. This is exactly a Byzantine failure. Otherwise (i.e., when $\boldsymbol{f} = 0$), the behavior is the same as in Figure 11(a).

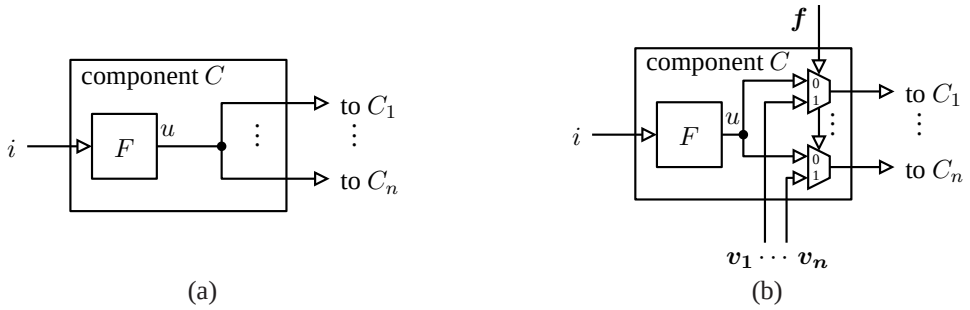

Fig. 11. (a) A non-faulty component C connected to three other components; (b) The same component subject to Byzantine failures.

6 Advanced DCS features

There are several other features offered by our method, which we present in this section. The first one concerns shared resources (for instance a communication device): we will show that, just by modifying the DCS objective that specifies the desired behavior, the user can switch from a resource in shared access to a resource in mutual exclusion (Section 6.1). The second feature is optimal DCS, which allows us to enforce quantitative constraints on the synthesized fault tolerant system. Such constraints are very useful when designing embedded systems, e.g., power consumption, memory footprint, bandwidth, and so on (Section 6.2).

The remaining features concern the technical aspects of DCS. First we show how conditioned DCS objectives can be used to take degraded modes explicitly into account within the fault tolerant policy (Section 6.3). Secondly, we show how synchronous observers can be used to refine the fault tolerant objectives and to express more complex objectives (Section 6.4).

6.1 Shared resources

We consider two tasks τ_1 and τ_2, running on the same processor, and competing for a shared resource, for instance a communication device connected to their processor. Figure 12 shows the LTSs of both tasks: each starts in state X_i, where it does not have access to the resource, and goes to state A_i, where it has access to the resource, upon receiving event a_i.

Fig. 12. (a) LTS of task τ_1; (b) LTS of task τ_2.

In order to specify the access policy of the shared resource, we design the DCS objective of the desired system in the following way:

- Shared access (default access policy): $true$.
- Mutual exclusion (useful when a resource can only be used by one client at a time): $S' = make\_invariant\,(S, \neg(A_1 \wedge A_2))$.
- Continuous access (useful when a resource must be monitored or controlled at all time, for instance a robotic arm because of the gravity compensation that must be constantly applied to prevent it from falling): $S' = make\_invariant\,(S, A_1 \vee A_2)$.

The above formulas can be straightforwardly generalized to more than two tasks and more than one shared resource.

An interesting particular case of mutual exclusion is critical sections. Figure 13 shows the generic LTS of a task having n successive steps S_1 to S_n: when in step S_i, it must wait for the arrival of event t_i before going to the next step S_{i+1}; after the final steps S_n, it goes to the terminated step T. Then, a task τ_1 having n

successive steps and competing for a shared resource will be specified by putting in parallel the two LTSs of Figures 12(a) and 13: because of the parallel composition, its states will be pairs (s_1, q_1) with $s_1 \in \{X_1, A_1\}$ and $q_1 \in \{S_{1,1}, \ldots, S_{1,n}, T_1\}$ (with a renaming of the states S_i of the LTS of Figure 13 into $S_{1,i}$). Similarly, a second task τ_2 will be specified by putting in parallel the two LTSs of Figures 12(b) and 13 (with a similar renaming of the states). Then, the following function guarantees that only step S_i is a critical section for both tasks τ_1 and τ_2:

$$S' = make\_invariant\left(S, \neg\big((A_1, S_{1,i}) \wedge (A_2, S_{2,i})\big)\right)$$

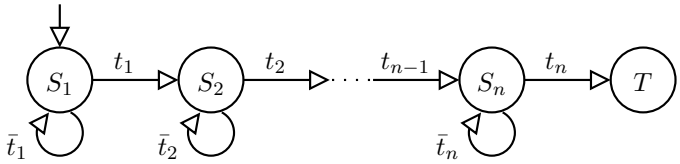

Fig. 13. LTS of a task having n successive steps S_1 to S_n.

6.2 Optimal discrete controller synthesis

It is possible to associate, to each transition and/or state of the initial system a **weight**, and to specify some **combination function** of the weights. This function is then used for the computation of the synchronous product, and it can be required that it never goes above or below some fixed maximal or minimal bound, or even that it be maximized or minimized. This is what **optimal synthesis** does [36,53,50,41,42]. Such an optimization can apply to single transitions [44] or to finite paths. Let us note that optimal synthesis does not guarantee that the controlled system will be deterministic, but only that it will be the most permissive one optimizing the combination function. It is indeed possible that two outgoing and controllable transitions produce the same result on the combination function.

Within our framework, it is very useful to model limited resources, like memory or power, which are crucial for embedded systems, and we have demonstrated its applicability in two case studies [27,22], the former is a single transition optimization, while the latter is a finite path optimization. Concretely, we use following additional DCS function, where ψ is any cost function from the states to the integers:

– $S' = maximize\_step\,(S, \psi)$ is a function that synthesizes and returns a controllable system S' such that, from any state q_i, all the controllable outgoing transitions that lead to successor states q_{i+1} having a non maximal cost function $\psi(q_{i+1})$ are disabled. Optimization must always be applied after the invariance and reachability objectives, as a means of choosing one optimal solution among the correct ones.

For instance, we can specify three tasks τ_1, τ_2, and τ_3, each modeled by the LTS of Figure 10, running on three processors P_1, P_2, and P_3. Table 1 gives the power consumption costs and the quality of each task onto each processor, as well as the maximal power consumption of each processor. The notion of quality refers, e.g., to computation tasks that can give more accurate results if given better computing resources. The combination function for both weights is the sum, that is, the cumulative power consumption (resp. the cumulative quality) is the sum of the power consumptions (resp. of the qualities) in all the active states of the synchronous product.

		power consumption C_i^j per task and processor			quality Q_i^j per task and processor		
		P_1	P_2	P_3	P_1	P_2	P_3
task	τ_1	4	4	2	3	5	3
	τ_2	2	2	3	2	2	5
	τ_3	2	3	4	2	2	5
	bound b_i	5	3	6			

Table 1. Power consumption C_i^j and quality Q_i^j of the tasks τ_j on the processors P_i, with the bounds b_i giving the maximal power consumption of each processor.

The complete system specification can therefore be given by the LTSs of Figure 10 for the three tasks, of Figure 5(a) for the three processors, and of Figure 6(a) for the environment model. A **configuration** of the system is

an assignment of the tasks to the processors: for instance, the configuration $\langle A_1^1|A_2^2|A_3^3\rangle$ indicates that the tasks τ_1, τ_2, and τ_3 are respectively executed on the processors P_1, P_2, and P_3, while the configuration $\langle A_1^2, A_1^3|\emptyset|A_3^1\rangle$ indicates that τ_1 is on P_3, that τ_2 and τ_3 are on P_1, and that P_2 executes no task. There is a total of 27 different configurations.

A basic fault tolerant policy can require that no task be active on a faulty processor:

$$S' = make\_invariant\left(S, \neg \bigvee_{j=1}^{3} \bigvee_{i=1}^{3} (A_i^j \wedge ERR_i)\right)$$

Thanks to optimal DCS, it is possible to refine this policy by requiring that no processor exceeds its maximal power consumption bound:

$$S'' = make\_invariant\left(S', \bigwedge_{i=1}^{3} \left(\sum_{j=1}^{3} C_i^j \leq b_i\right)\right)$$

Note that, in the above objective, the predicate φ that specifies the subset E is actually a *constraint on the costs* of the component states; this is a generalization of the state predicates of Equation (3) in Section 3.3.

Finally, again thanks to optimal DCS, we can require that the cumulative quality of the tasks be maximal:

$$S''' = maximize\_step\,(S'', Q_g)$$

where the quality Q_g of the current global state of the system is the sum of qualities of the tasks, each in its current state (A_1^j, A_2^j, or A_3^j): $Q_g = \sum_{j=1}^{3} Q^j$, where Q^j has the value of Q in the current state of task τ_j.

For instance, suppose that P_2 fails (i.e., the uncontrollable event e_2 occurs) while the system is in the configuration $\langle A_1^1|A_2^2|A_3^3\rangle$. From this state, the 27 possible configurations are reachable. We shall not discuss all of them, but rather just explain that the three following configurations must be avoided in the controlled system since they violate one of the required properties:

- $\langle A_1^1|A_2^2|A_3^3\rangle$, after no migration, violates the fault tolerance property;
- $\langle A_1^1, A_1^2|\emptyset|A_3^3\rangle$, after the migration of τ_2 to P_1, violates the maximal power consumption property because of the bound b_1 of P_1;
- and $\langle A_1^1|\emptyset|A_3^2, A_3^3\rangle$, after the migration of τ_2 to P_3, violates the maximal power consumption property because of the bound b_3 of P_3.

As a consequence, the three corresponding transitions must be disabled by the synthesized controller. In contrast, the two following configurations satisfy the four required properties:

- $\langle A_1^2, A_1^3|\emptyset|A_3^1\rangle$, after the migration of τ_1 to P_3 and of τ_2 and τ_3 to P_1;
- and $\langle A_1^2|\emptyset|A_3^1, A_3^3\rangle$, after the migration of τ_1 to P_3 and of τ_2 to P_1.

Among those two configurations, thanks to the quality property, the controller should prefer $\langle A_1^2|\emptyset|A_3^1, A_3^3\rangle$ since the corresponding cumulative quality is 10 ($3 + 2 + 5$), instead of only 7 ($2 + 2 + 3$) for $\langle A_1^2, A_1^3|\emptyset|A_3^1\rangle$.

What we have just described involves an optimization of the controlled system only over a single step. That is, from the current state, it selects the transition that leads to the immediate successor state that optimizes the given criteria. But in general, it does not select the transition that leads ultimately to the reachable state that optimizes the given criteria. This is known as path optimization, and we have demonstrated its utility within our framework of DCS for fault tolerance [22].

Optimal DCS over finite paths involves a modified Bellman DCS algorithm [5] in order to deal with path having infinite loops, something that the classical Bellman algorithm cannot do. We will not go into the details of our modified algorithm, which can be found in [22]. We illustrate its application to fault tolerance on tasks having several successive phases separated by checkpoints. Such a task is exemplified by the LTS of Figure 14(a): the task, named τ_1, begins in the idle state I^1, then goes in the ready state R^1, before starting its first phase A on either one of the processors P_i (in one of the states A_i^1); the task can be migrated to another processor while still in its first phase (these are the "horizontal" transitions which are rollbacks); only after passing its first checkpoint can it start its second phase B, again on either one of the processors P_i (in one of the states B_i^1); again, the task can be migrated to another processor while still in its second phase; finally, only after passing its second checkpoint can the task terminate and go to the terminated state T^1. In this LTS, there are only two phases / checkpoints, but in

the general case this number is arbitrary. Like in Section 5.1, the events $\boldsymbol{r^1}$ and $\boldsymbol{t^1}$ (the latter playing the role of the second checkpoint events) are uncontrollable; the first checkpoint events $\boldsymbol{c^1}$ are also uncontrollable (and similarly for the LTS of Figure 14(b)).

Furthermore, we associate to each state a static cost representing the cost for the task to traverse this state. We also assume that the processors are subject to permanent crash failures, and we adopt the environment model of Figure 6(a). The problem consist now in finding a controller that will guarantee that no task be active on a faulty processor (fault tolerance policy), and that the total costs of executing the tasks from their idle to their terminated state be minimized; this cost is computed by summing, for each task, the individual cost of each traversed state. For instance, if a task is migrated from P_1 to P_2 before its first checkpoint, then it must pay the cost of its first phase twice: this is consistent with the classical notion of checkpoints and rollback.

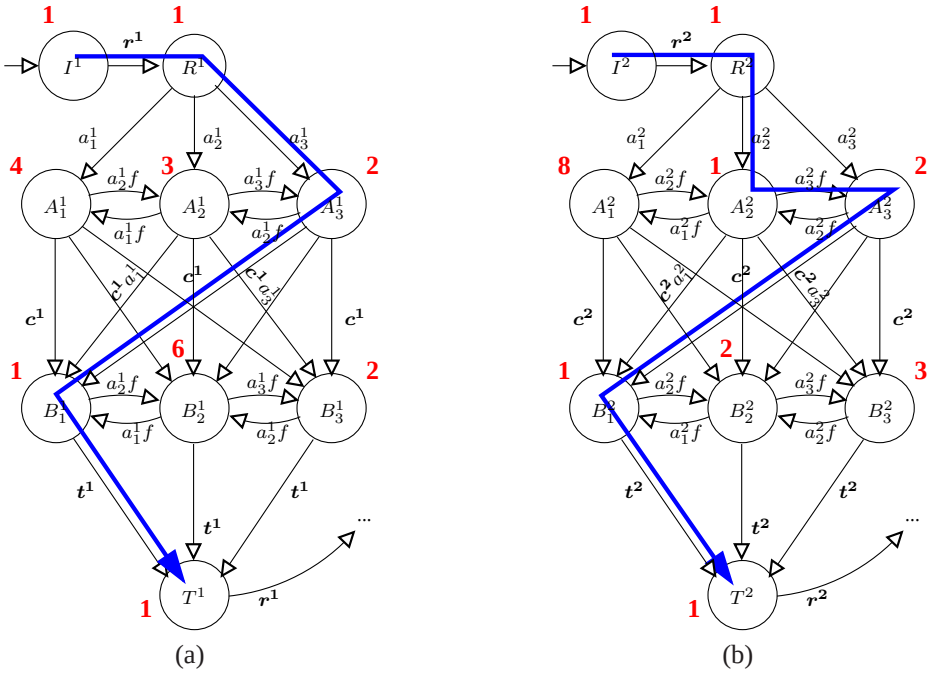

Fig. 14. Example of runs for two tasks, T^1 (a) and T^2 (b), executing on three processors, when P_2 fails.

Figures 14(a) and 14(b) show the examples of runs for two tasks, respectively named T^1 and T^2, executing on three processors. The cost of each state is indicated next to its state. 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. First, 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; in the meantime, 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 $\boldsymbol{c^{1,2}}$. Just after a checkpoint, processor migrations can also occur for optimality reasons: here, both T^1 and T^2 migrate respectively from P_3 to P_1 in order to achieve their best execution cost.

6.3 Conditioned discrete controller objectives

Conditioned DCS objectives are very useful to address **degraded modes** of control, e.g., to achieve a management of the degraded modes. The principle is that each such mode is specified by, on one hand a predicate on states $(\varphi_i)_{1\leq i\leq n}$, and on the other hand a condition $(C_i)_{1\leq i\leq n}$. Then, there are two possibilities, either a conditioned invariance or a conditioned reachability:

– A single conditioned invariance objective is achieved by $S' = \mathit{make\_invariant}\,(S, C_i \Rightarrow \varphi_i)$, where the predicate $C_i \Rightarrow \varphi_i$ is of course equal to $\neg C_i \vee \varphi_i$. In the case of multiple conditioned invariance objectives, the controlled system is synthesized by $S' = \mathit{make\_invariant}\,(S, \bigwedge_{i=1}^{n}(C_i \Rightarrow \varphi_i))$. A useful instantiation of this is the synthesis function $S' = \mathit{make\_invariant}\,(S, (C \Rightarrow \varphi_1) \wedge (\neg C \Rightarrow \varphi_2))$, which amounts to switch from the DCS objective φ_1 to φ_2 and back according to the condition C.

– A conditioned reachability objective is a bit more elaborate, and requires first to transform the objective into an invariance one: $keep\_reachable\,(S, E) = make\_invariant\,(S, reachable\_under\_control\,(S, E))$ (see Section 2.2). Then, for a single conditioned objective, the controlled system is synthesized by $S' = make\_invariant\,(S, C_i \Rightarrow reachable\_under\_control\,(S, \varphi_i))$. In the case of multiple conditioned reachability objectives, the controlled system is synthesized by $S' = make\_invariant\,(S, \bigwedge_{i=1}^{n}(C_i \Rightarrow reachable\_under\_control\,(S, \varphi_i)))$.

As a consequence, the obtained controlled system will satisfy the predicate φ_i (that is, either the subset of states satisfying φ will be invariant or reachable) provided that the condition C_i holds. Of course, this technique involving conditioned synthesis objectives can be used to design many other systems and not just degraded modes. A full case study involving degraded modes and conditioned DCS objectives is presented in Section 7.1.

6.4 Synchronous observers

Observability is an important notion in discrete controller synthesis. Just like the alphabet $\mathcal{I}$ of the language $\mathcal{U}$ is partitioned into two subsets (the set $\mathcal{I}_c$ of controllable events and the set $\mathcal{I}_u$ of uncontrollable events), it can also be partitioned into two other subsets: the set $\mathcal{I}_o$ of **observable** events and the set $\mathcal{I}_{uo}$ of **unobservable** events. The idea is that the controller must behave in the same way whether an unobservable occurs or not [39,18].

Within our framework, it is sometimes useful to express a synthesis objective that refers to the output of one of the system's LTSs. In such a case, if its internal states of this LTS were observable, then the controller could make some of its internal states unreachable (by disabling incoming transitions). In contrast, if this LTS's internal states were unobservable, then the controller would not be able to make them unreachable. As we can see, this is a different notion of observability as the one defined in [39,18], since it refers to the states rather than to the events and transitions.

A typical situation where a synchronous observer is useful is a system consisting of a plant coupled to one or more sensors (and with a model of the fault hypothesis as usual). The purpose of such a design is to take into account the value failures of the sensors. However, it is not possible to express the synthesis objective w.r.t. the state of the sensors, because it directly tells whether or not the sensors are faulty, and this contradicts our objective to tolerate the value failures of the sensors, hence without knowing if the sensors are faulty or not. For this reason, the synthesis objective must be expressed w.r.t. the state of the plant. But in this case, the possible value failure of the sensors will not be taken into account when synthesizing the controller, because the closed loop consisting of the plant and the controller is independent of the values of the sensors.

This situation is illustrated by Figure 15: the synthesized controller interacts only with the plant, independently of the eventual value failures of the sensors.

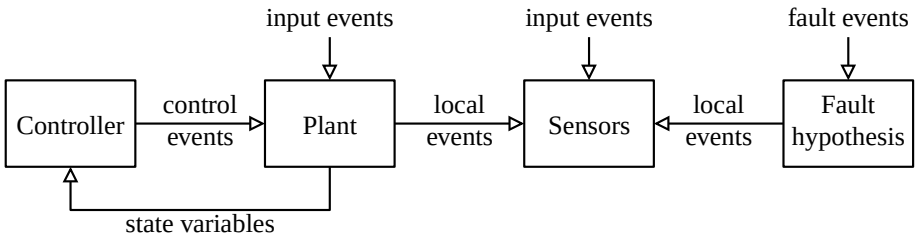

Fig. 15. A controlled system equipped with sensors subject to failures.

Therefore, we propose to add a **synchronous observer** [29] to the system, an LTS whose job is to observe the outputs of the sensors, and to go in a state named "BAD" as soon as these outputs correspond to the former synthesis objective. As a consequence, the new synthesis objective becomes $\neg BAD$, that is, the BAD state should be unreachable. Now the closed loop includes the plant, the controller, *and* the sensors. This new situation is illustrated by Figure 16.

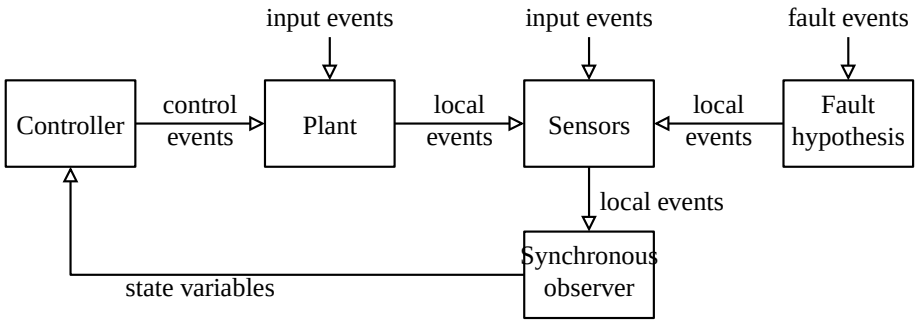

Fig. 16. The same system as in Figure 15 with a synchronous observer.

We have conducted with Huafeng Yu a full case study, consisting of a water tank with liquid level sensors, subject to value failures [28].

7 Case studies

Throughout Sections 4 to 6, we have used numerous examples extracted from three previously published case studies [27,23,28,22]. These examples show the usefulness of our framework. To further demonstrate this, we detail in this section two unpublished case studies: a system tolerant to the failures of its actuators, and the Byzantine generals revisited.

7.1 A system tolerant to the failures of its actuators with conditioned synthesis objectives

The system under study in this section consists of two tanks of liquid, connected by two pipes; it is a benchmark defined by the COSY group ("Control of Complex SYstems") of ESF ("European Science Foundation"). The left tank can be filled with liquid by opening the valve V_0. The right tank can be emptied by opening the valve V_3. The flow in the upper (resp. lower) pipe is controlled by the valve V_1 (resp. V_2). Each valve is subject to fail silent faults; in other words, it is the **actuators** of the system that can fail. This is illustrated in Figure 17. The level of the liquid in the left tank is abstracted as either N_0 (the tank is empty), N_1 (the level is between the lower and the upper pipe), N_2 (the level is above the upper pipe), or N_3 (the tank is over-flooding). Similarly, the level in the right tank is abstracted as either N'_0, N'_1, N'_2, or N'_3.

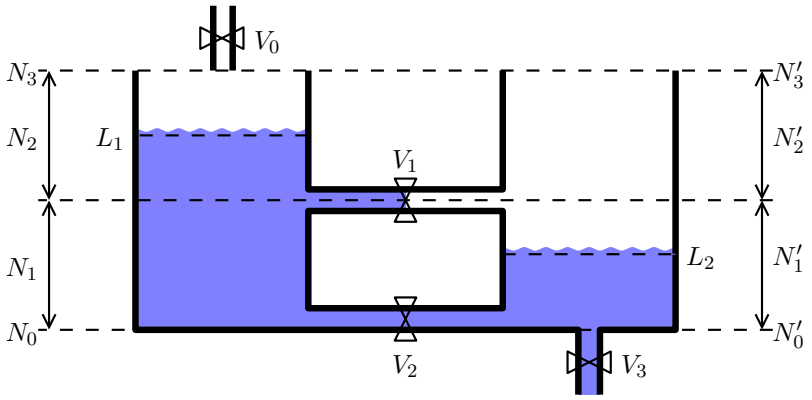

Fig. 17. A system with two tanks, two pipes and four valves.

We can observe that this system offers only a limited form of redundancy, since each actuator (valve) plays a specific role that cannot be directly fulfilled by the other actuators. For this kind of systems, it is always difficult to elaborate efficient fault tolerance strategies. We will see however that DCS brings satisfactory results to this respect.

Each valve V_i is an actuator subject to failure, and can hence be modeled by the LTS of Figure 18; this LTS is similar to what has been shown in Figure 7(a). When not faulty, the valve is either open (state O_i) or closed (state C_i), and can switch from the open to the close state according to the controllable input c_i, or vive versa

with o_i. The valve becomes faulty following the event $\boldsymbol{f_i}$, and goes either in the FO_i state if it was open at the time of the failure, or to the FC_i state if it was closed. The failures are permanent, which is modeled by the fact that the states FO_i and FC_i are sink states. The LTS of valve V_i also outputs a variable named v_i that represents the status of the valve: by convention, 1 means open while 0 means closed.

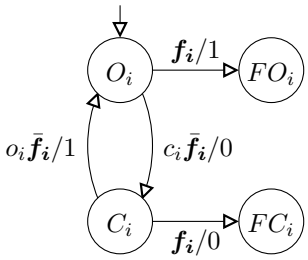

Fig. 18. LTS modeling the behavior of valve V_i.

For the sake of simplicity, we assume that all the valves have exactly the same flow per time unit. For instance, when V_0, V_2 and V_3 are open while V_1 is closed, the level in both tanks does not change. Also, because of the rules of communicating tanks, the level in the right tank can never be greater than the level in the left tank. The behaviors of the left and right tanks being tightly interdependent one of another, it is not possible to model each as a separate LTS. Rather, we propose the LTS of Figure 19 to model the joint behavior of the two tanks according the status of the four valves.

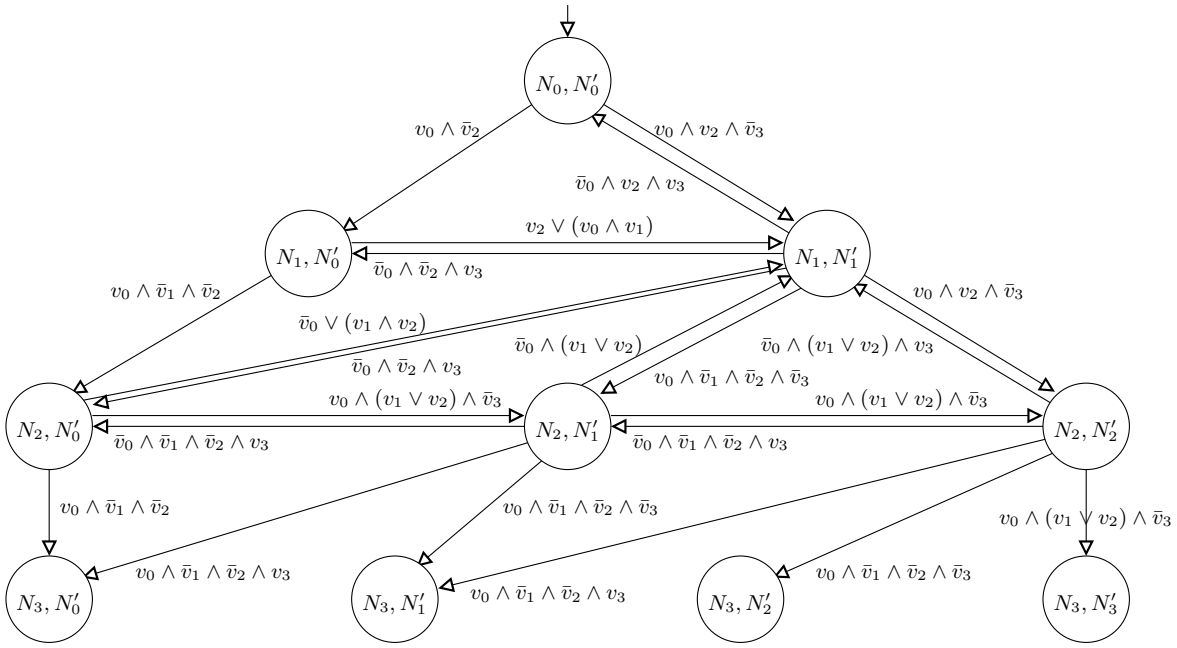

Fig. 19. The LTS modeling the joint behavior of the two tanks.

This model assumes that, if the right tank is empty (state N_i, N_0'), then it is not possible to empty the left tank without temporarily filling the right tank. In other words, transitions from N_i, N_0' to N_{i-1}, N_0' are forbidden. Furthermore, this model forbids two level changes in a row: for instance, to move from N_0, N_0' to N_2, N_0', the system must before go into N_1, N_0'. Finally, to avoid too complex drawings, the self-transitions that make the LTS reactive have been omitted in Figure 19.

The plant is therefore the synchronous product of the LTSs of the double tanks (Figure 19) and of the four valves (Figure 18). It is show in Figure 20.

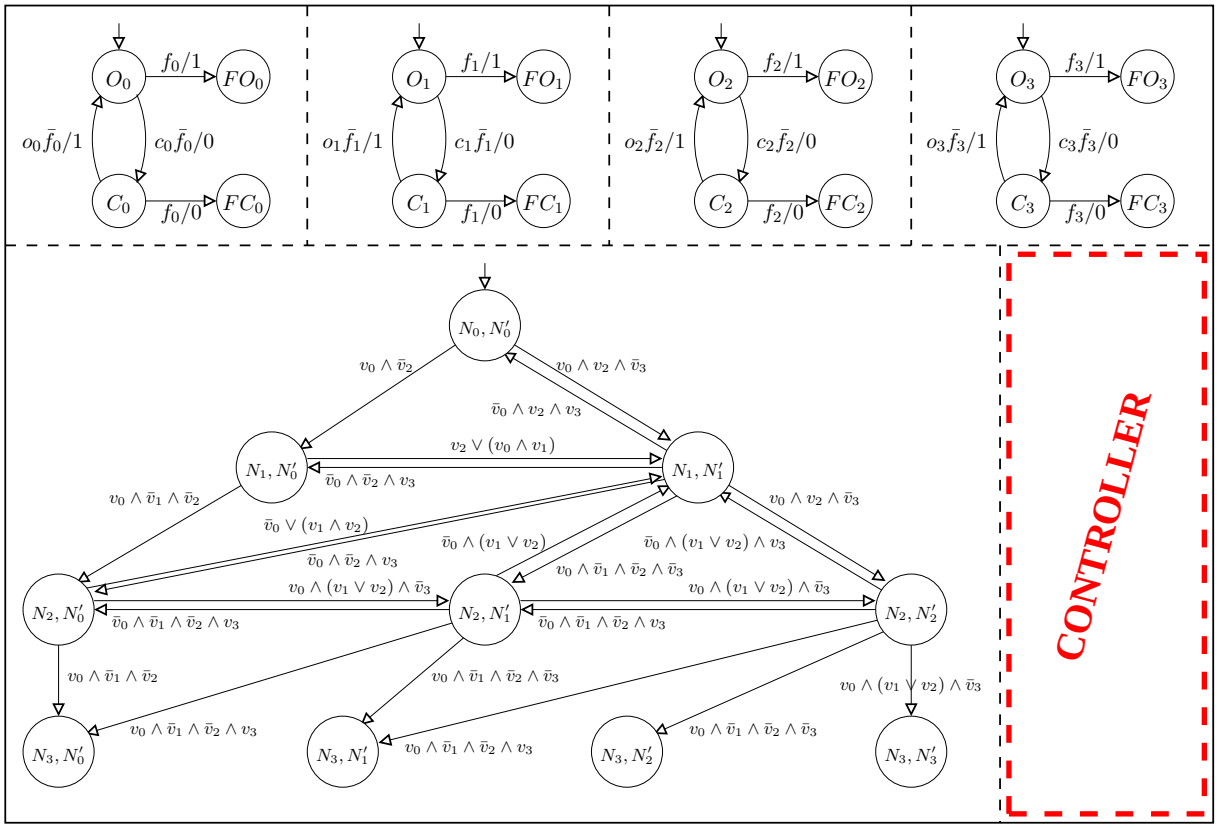

Fig. 20. Complete system made of the tanks and the four valves.

The problem we want to address for our two-tanks system is to synthesize a controller guaranteeing the two following functions:

LV1: No tanks over-floods. This is achieved by $S' = make\_invariant\left(S, \neg \bigvee_{i=0}^{3} N_3 \vee N'_i\right)$. Note that, because of the physical configuration of the two tanks and the two pipes, the right tank can over-flood only if the left tank does so too. In other words, it suffices to prevent the system from going in the state N_3, instead of $N_3 \vee N'_3$.

LV2: The level in the left and right tanks must be regulated respectively at N_2 and N'_1. This is achieved by $S'' = keep\_reachable\,(S', \{(N_2, N'_1)\})$.

Unfortunately, if the valve V_0 fails while it is open, and at the same time either the valve V_3 fails while it is closed, or both valves V_1 and V_2 fail while they are closed, then the objective **LV1** becomes impossible to satisfy. This intuition is confirmed by SIGALI that fails to synthesize the required controller. The fundamental reason for this is the low level of redundancy offered by the fault intolerant system.

One solution would be to strengthen the fault hypothesis, by assuming that only one valve can fail. In terms of our system model, this would result in adding an LTS similar to the one of Figure 6(a). Rather, we choose to **condition the synthesis objective** according to the faults of the valves. Our synthesis functions **LV1** and **LV2** therefore become:

LV1': If the valve V_0 is not stuck in the faulty and open state, then no tank over-floods. This is achieved by $S' = make\_invariant\left(S, \neg FO_0 \Rightarrow \neg \bigvee_{i=0}^{3}(N_3, N'_i)\right)$.

LV2': If the four valves work fine, then the level in the left and right tanks must be regulated respectively at N_2 and N'_1. This is achieved by (see Section 6.3 for the explanations about conditioned reachability objectives) $S' = make\_invariant\left(S, \bigwedge_{i=0}^{3}(O_i \vee C_i) \Rightarrow reachable\_under\_control\,(S, \{(N_2, N'_1)\})\right)$.

This two-tanks system has been implemented in MATOU by Safouan Taha [51]. The controlled system synthesized by SIGALI with the objectives **LV1'** and **LV2'** behaves as expected: while the four valves work fine, it regulated the liquid level at N_2 in the left tank and at N'_1 in the right tank.

We believe that this approach (DSC with conditioned synthesis objectives) is very useful to design fault tolerant systems. It guarantees by construction a specified level of fault tolerance, and offers a very elegant way to specify degraded modes in a system.

7.2 The Byzantine generals revisited

In this section, our goal is to model Byzantine faults by means of LTSs and uncontrollable events, and to obtain with DCS the same result as Lamport et al. [37] in the particular case of 4 generals. Note that the result in [37] is parametric for n generals and hence much more general than what we achieve here. It would be interesting in future work to extend our DCS framework to handle such parametric models.

In [37], Lamport et al. define the **Byzantine generals problem** in the following way: n divisions of the Byzantine army, each commanded by its own general, are camped outside an enemy city. The generals must decide on a common plan of action, either **attack** or **retreat**, by communicating with one another only by oral messages. The problem is that some generals are **traitors** who try to prevent the loyal generals from reaching agreement. One of the generals is the **commander** of the army, while the $n-1$ remaining ones are his **lieutenants**.

The commander first sends an order (attack or retreat) to his $n-1$ lieutenants. If he is loyal, then he must send the same order to all his lieutenants; but if he is a traitor, then he can send different orders to his lieutenants, that is, incoherent orders. It was after this article that incoherent value faults have been called **Byzantine faults**.

Then, each lieutenant transmits the received order to all the other lieutenants. Again, a loyal lieutenant must transmit the order he has received from his commander to all the other lieutenants, but not if he is a traitor.

The goal is to find an algorithm guaranteeing that the loyal generals will reach a **consensus** for their plan of action. Formally, the two following interactive consistency conditions must be satisfied:

IC1: All the loyal lieutenants obey the same order.
IC2: If the commander is loyal, then each loyal lieutenant obeys the order sent to him.

The algorithms 1.1 and 1.2, proposed by Lamport et al., are respectively the actions performed by the commander and by the $n-1$ lieutenants: m is the number of potential traitors (the actual number of traitors is not known), v is the initial order, and i is the number of the lieutenant.

Algorithm 1.1 Byzantine Commander (m, v)
1 Send my order v to the $n-1$ lieutenants;

Algorithm 1.2 Byzantine Lieutenant (m, i)
1 v_i := value received from the commander;
2 **if** $m = 0$ **then**
3 Use as order the value v_i;
4 **else**
5 Send v_i to the $n-2$ other lieutenants;
6 **forall** $j \neq i$ **do**
7 v_j := value received from the lieutenant j;
8 **end do**
9 Use as order the majority $maj\,(v_1, v_2, \ldots, v_{n-1})$;
10 **end if**

Lamport et al. demonstrate by induction over m that, to accommodate the presence of at most m traitors, there must be at least $3m+1$ generals to guarantee that all the loyal generals reach the consensus. Three hypotheses on the exchanged messages are necessary: each message sent is correctly received, the receiver knows the sender, and the absence of a message can be detected. In terms of computer science, these hypotheses can be satisfied by a fully connected point-to-point network, and with synchronized clocks. Moreover, the function $maj\,(v_1, v_2, \ldots, v_{n-1})$ is such that if a majority of the values v_i is equal to v, then the result is v.

The problem we want to address in this section is the following: among the n Byzantine generals, how many at most can be traitors? In order to answer to this question with DCS, we model the environment as the *most permissive possible* LTS, having as inputs e_c, e_1, e_2, and e_3 (respectively the betrayal of the commander and the three lieutenants), and producing as outputs t_c, t_1, t_2, and t_3 (each of those will be used as input respectively by the LTS of the commander and the three lieutenants; see Figure 21). We note Loy_c and Tra_c respectively the state where the commander is loyal or traitor, and Loy_i and Tra_i the state where the i-th lieutenant is loyal or traitor. The following LTS is therefore the most permissive environment model:

$$\left\langle Loy_c \xrightarrow{e_c/t_c} Tra_c \right\rangle \;||\; \left\langle Loy_1 \xrightarrow{e_1/t_1} Tra_1 \right\rangle \;||\; \left\langle Loy_2 \xrightarrow{e_2/t_2} Tra_2 \right\rangle \;||\; \left\langle Loy_3 \xrightarrow{e_3/t_3} Tra_3 \right\rangle$$

Our idea is that DCS will *constrain* this environment model by *preventing* some generals to be traitors, that is by *inhibiting* some of its transitions. In contrast with Section 4.1, the events e_c, e_1, e_2, and e_3 must therefore be *controllable*. In other words, we want to obtain by DCS the most permissive LTS guaranteeing that the generals will reach the consensus in any circumstances. Note that when doing so, we limit in fact the *actual* number of traitors instead of the *potential* number of traitors. In other words, our model considers that m is the *actual* number of traitors instead of the *potential* number of traitors.

We note Att_c and $Retr_c$ respectively the state where the commander attacks or retreats, and Att_i and $Retr_i$ the state where the i-th lieutenant attacks or retreats.

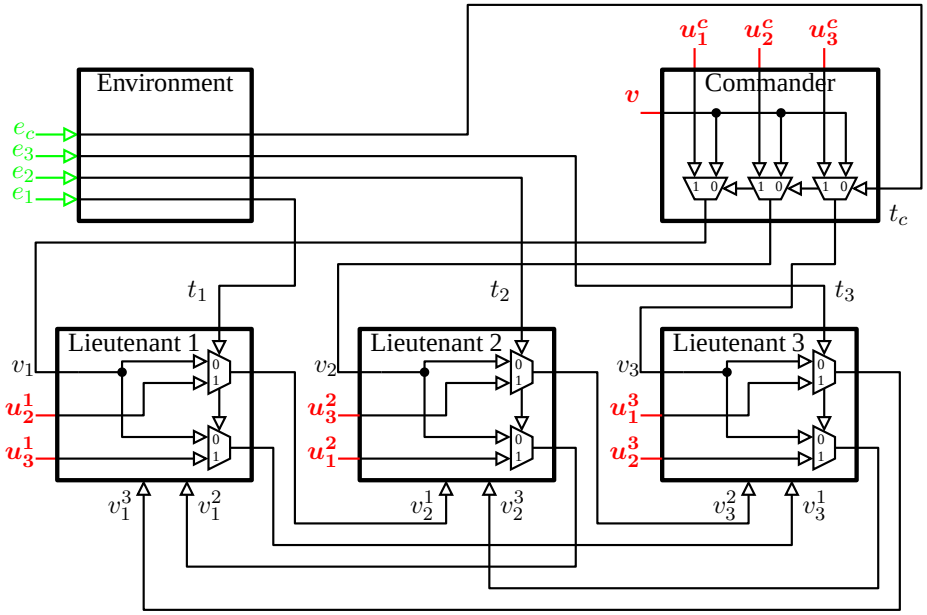

Fig. 21. Complete system made of the environment model, the commander, and the three lieutenants.

The LTS of the commander receives, as input, the initial order $\boldsymbol{v}$ he is supposed to send to the three lieutenants ($\boldsymbol{v}$ being an uncontrollable event): $\boldsymbol{v} = true$ means attack while $\boldsymbol{v} = false$ means retreat. To model the fact that, if he is a traitor, then he can send incoherent messages, we add three uncontrollable inputs, $\boldsymbol{u_1^c}$, $\boldsymbol{u_2^c}$, and $\boldsymbol{u_3^c}$ (as we have shown in Section 5.3 and in Figure 11). In his normal mode of operation, his three outputs are equal to $\boldsymbol{v}$. But when he is a traitor, his three outputs are each equal to one of his three uncontrollable inputs. Similarly, the LTS of lieutenant i receives, as input, the order v_i sent by the commander, and which he is supposed to transmit to the two other lieutenants, via his outputs v_2^i and v_3^i. To model the fact that, if he is a traitor, then he can send incoherent messages, we add two uncontrollable inputs, $\boldsymbol{u_2^i}$ and $\boldsymbol{u_3^i}$. Finally, the LTS of lieutenant i must compute the majority of the three received values, v_i, v_i^2 and v_i^3, in order to determine if he must go to the state Att_i or $Retr_i$.

In terms of DCS, the properties **IC1** and **IC2** translate into:

– Property **IC1** = unreachability of the states such that the predicate $\forall i \neq j, Loy_i \wedge Loy_j \wedge \big((Att_i \wedge Retr_j) \vee (Retr_i \wedge Att_j)\big)$ is true, that is:

$$S' = make\_invariant\ \big(S, \forall i \neq j, Loy_i \wedge Loy_j \wedge \big((Att_i \wedge Retr_j) \vee (Retr_i \wedge Att_j)\big) = false\big)$$

– Property **IC2** = unreachability of the states such that the predicate $\forall i, Loy_c \wedge Loy_i \wedge \big((Att_c \wedge Retr_i) \vee (Retr_c \wedge Att_i)\big)$ is true, that is:

$$S' = make\_invariant\ \big(S, \forall i, Loy_c \wedge Loy_i \wedge \big((Att_c \wedge Retr_i) \vee (Retr_c \wedge Att_i)\big) = false\big)$$

With Nour Brinis we have implemented the system of Figure 21 in MATOU. By asking to SIGALI to synthesize a controller with the two properties above, we have obtained a controlled system of four Byzantine generals tolerating the presence of one traitor among them [12]. This result is consistent with the theorem of Lamport et al. The originality of our approach lies in the usage of uncontrollable inputs to model incoherent values, as well as in the usage of DCS to determine the maximal number of Byzantine admissible faults, by producing the environment model that is the most permissive and still guarantees that the generals reach the consensus whatever the circumstances.

8 Related work

To the best of our knowledge, although they do not mention any software implementation, Cho and Lim have been the first ones to develop the idea of making a system fault tolerant thanks to DCS, by considering faults as uncontrollable events [17]. Their results are based on the framework of supervisory control of discrete event systems of Ramadge and Wonham [48]. First, the set of events Σ is partitioned into $\Sigma = \Sigma_c \cup \Sigma_{uc} = \Sigma_n \cup \Sigma_{an}$, respectively the subsets of controllable, uncontrollable, normal, and abnormal events; moreover, $\Sigma_{an} \subset \Sigma_{uc}$. With respect to a marker set of states Q_m (the objective for the control), they define a **recurrent event** to be such that Q_m can be reached from its originating state, either through controllable or other recurrent events. Then, a **fault event** is an abnormal event that does not prevent the system from reaching Q_m, otherwise it is a **failure event**. Their guideline is that a fault is a malfunction while a failure is a total breakdown. Finally, a system is **fault tolerant** w.r.t. Q_m if, when any abnormal event occurs during the execution, either there must exist another event sequence which can reach Q_m, or the path to this abnormal event can be eliminated. Any event sequence, which consists of normal events or fault events and which drives the initial state to Q_m, is called a **tolerant fault event sequence** (TFES) if, for each normal event, all the possible events following the corresponding states are either controllable or other recurrent events. The set of all TFES is then taken as the legal language K, which is achievable by construction, i.e., both controllable and observable. Finally, the plant is constructed as the parallel composition of several finite state automata. The differences w.r.t. our own work reside in:

- their usage of a set of states as control objective instead of invariance or reachability properties, which are easier to use in practice, all the more when in conjunction with a synchronous observer;
- their usage of the basic parallel product instead of the synchronous product: the latter limits the combinatorial explosion, without avoiding it entirely though;
- the absence of clear definition of their fault hypothesis, while we model it as an LTS, which is both more formal and more flexible;
- our usage of optimal DCS that provides more possibilities of synthesis as well as to limit the non-determinism of the controlled system;
- finally, our usage of a DCS software tool while Cho and Lim do not mention such a tool.

In [45], Marchand and Samaan exemplify the use of DCS in the specific case of a power transformer. Like them, we model failure events with uncontrollable boolean inputs. Their modeling is very specific to their case study: for instance, the fault propagation is influenced by the opening and closing of circuit-breakers. In contrast, our framework covers a much wider range of fault tolerance issues.

The technique proposed by Kulkarni and Arora in [33], and improved in [34,35,9], is close to our own work. It involves synthesizing automatically a fault tolerant program starting from an initial fault intolerant program. In their model, a program is a set of processes, each with its local variables. Each program's state is a valuation of the program's variables. Two execution models are considered: the **high atomicity** model, where the program can read and write any number of its variables in one atomic step (i.e., it can make a transition from any one state to any other state), and the **low atomicity** model, where it can not (actually, each process can write only its own variables, and can read only its own variables and its neighbor's). The initial fault intolerant program ensures that its specification is satisfied in the absence of faults, but no guarantees are provided in the presence of faults. Then, a fault is a subset of the set of transitions. The authors consider three levels of fault tolerance:

- the **failsafe ft**: even in the presence of faults, the synthesized program guarantees safety;
- the **non-masking ft**: even in the presence of faults, the synthesized program recovers to states from where its safety and liveness are satisfied;
- and the **masking ft**: conjunction of the two above mentioned levels.

To address their two models of atomicity and their three levels of fault tolerance, the authors propose a sound and complete algorithm that is polynomial in the state space of the source fault intolerant program for the high atomicity model (resp. exponential for the low atomicity model). In the low atomicity model, the transformation problem is NP-complete, except for non-masking ft for which the complexity is unknown. Each transformation involves recursively removing bad transitions. However, some transitions cannot be removed (like the uncontrollable transitions in DCS), but this is the case only of fault transitions (while in DCS, any event can be uncontrollable, not only faults). An efficient BDD-based algorithm has been presented in [8] and implemented in the SYCRAFT tool [10].[8]

[8] SYCRAFT: http://www.cse.msu.edu/~borzoo/sycraft

Attie et al. have also proposed an automatic synthesis method for fault tolerant programs [3]. In their approach, a system is a set of concurrent processes, each consisting of a directed graph, where states are connected by transitions labeled by guarded commands. At each execution step, one process is randomly chosen to fire an enabled transition from its current state. To specify such systems, the temporal logic CTL is used as a specification language. Such a specification allows to distinguish between the **safety part** and the **liveness part** of the system. Faults are modeled as guarded commands that perturb the system's state. The occurrence of a fault is modeled as a directed graph whose transitions are labeled by fault guarded commands. Attie et al. use the same fault tolerance properties as Kulkarni and Arora: in **masking tolerance**, both the safety and the liveness parts are respected; in **fail-safe tolerance**, only the safety part but not necessarily the liveness part is respected; and in **non-masking tolerance**, the liveness part is always respected but the safety part is only eventually respected.

The fault tolerance synthesis problem starts with a problem specification (a CTL formula of the form $init\_spec \wedge AG(global\_spec)$), a fault specification (a CTL formula F), a problem-fault coupling specification (a CTL formula $AG(coupling\_spec)$), and a type of tolerance TOL (either masking, fail-safe, or non-masking). The goal is to synthesize a concurrent program that satisfies $init\_spec \wedge AG(global\_spec)$ in the absence of faults, satisfies $AG(coupling\_spec)$ in the absence of faults, and is TOL-tolerant to F for $init\_spec \wedge AG(global\_spec)$. The authors use the decision procedure of [24] to solve this problem, i.e., to synthesize the recovery behavior that conforms to the required tolerance properties. The overall time complexity is exponential in the size of the specification (i.e., the size of the problem specification plus the size of the problem-fault coupling specification).

There are three important differences between our approach and the ones of Kulkarni and Arora and Attie et al. Firstly, their model of computation (MoC) is the non-deterministic interleaving, while ours is the synchronous deterministic parallel composition. We believe that, when designing safety critical distributed systems, a deterministic MoC is better suited than a non-deterministic one (it makes debugging easier and facilitates formal model-based methods). This claim is supported by the successes achieved by the synchronous MoC [6], in particular in avionics [11]. Secondly, only fault transitions are uncontrollable, while we use uncontrollable events to model fault events but also any event decided non-deterministically by the environment. And finally, our method can handle some form of optimality w.r.t. costs associated to states of the system.

Based on the work of Kulkarni and Arora, Gärtner and Jhumka propose a way to deal also with non fusion closed traces [26]. A specification is **fusion closed** iff the entire history of every trace is present in every state of the trace (hence the next state of the systems depends only on its current state and on the inputs, i.e., not on the sequence of previous events). The usual way to transform a non fusion closed specification into a fusion closed one involves adding **history variables** to the states, in order to remember the sequence of past inputs. However, in general this is exponential. The authors propose a polynomial method, which involves splitting fusion paths (here a new state is added), and then removing the bad fusion states. If n is the number of state of the initial non fusion closed specification, then, at worst, the number of states of the resulting equivalent fusion closed specification is $\mathcal{O}(n^2)$. This result has later been generalized in [35].

Kamach et al. have applied DCS to a system with several modes of operations [32]. Their approach allows the user to specify, for instance, one **nominal mode** and one **degraded mode** for a subsystem, and to switch this subsystem between those two modes according to two uncontrollable events, called **commutation events**. They present a case study consisting of a small industrial pneumatic production line, with two jack cylinders and one pump. The horizontal jack cylinder has a degraded mode, where it can no longer move. The commutation events associated to the horizontal jack cylinder are p (failure) and r (repair). This case study has been implemented with the TCT tool of Ramadge and Wonham.

There are also works in the domain of hardware synthesis, or scheduler synthesis, in co-design. Similarities with our work exist, at least in the informal statement of the problems: the use of discrete event dynamical systems as formal models of reactive systems, be it Petri nets or LTS, to synthesize sequences in the presence of constraints of different kinds, with controllable and uncontrollable inputs. Hardware synthesis is an elaborate optimization and constraints process. It can involve notions related to game theory. However, there are differences with our work, which can be hidden by similarities of vocabulary. For example Cortadella et al. distinguish uncontrollable and controllable inputs by the constraints on the moment when they can be read, the objective being to avoid blocking schedules [19]. They make the relation with the notions, in the synchronous language ESTEREL, of "signal" (uncontrollable) and "sensor" (controllable).

In contrast, we use the words controllable and uncontrollable in the different meaning of Ramadge and Wonham, which is very classically accepted in the community of supervisory control, also called Discrete Controller Synthesis [48]. There, the synthesis involves computing the constraint in the value of controllable variables of a system (and not the moment of their "reading"), as a function of uncontrollable values and current state, so that the paths that can be taken in the controlled LTS do respect the properties given as synthesis objectives, and this

whatever the values of uncontrollable variables (and not the moment of their "reading"). Controllable events are used to inhibit some behaviors, through a constraint on their value so that the transitions they are labeling cannot be taken by the controlled system. Controllable events are inputs of the uncontrolled system, but not of the final system once the controller is integrated. The control which is synthesized in supervisory control concerns the values, not the moments of reading operations. Another significant difference with the work of Cortadella et al. is that the questions of fault tolerance, which we are treating, do not seem to be explicitly addressed there.

Another research area close to DCS is **planning**, a technique that has emerged from artificial intelligence. Results on the automatic generation of fault tolerant plans have been obtained by Jensen et al. [31]. After defining the general problem of finding a n-fault tolerant plan, the authors concentrate on 1-fault tolerant planning and present two algorithms based on Ordered BDDs [14]. The main limitation of their results is that they tolerate only one fault, while our model can accommodate an arbitrary number of faults.

Timed Game-Automata (TGA) can also be used in a framework for automating the addition of fault-tolerance. Quite naturally, one player could be the environment producing the faults, while the other player could be the controller trying to keep the system in a subset of safe states (this subset being formally specified as a reachability or safety property). Although there exists an efficient on-the-fly algorithm to verify safety and reachability properties for TGAs [16], implemented in the UPPAAL-TIGA tool[9], this appealing idea has never yet been applied to fault-tolerance.

Formal approaches to the design of fault tolerant systems have mostly considered the problem of **formal verification**, in the context of process algebra [49,13,7]. 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. What distinguishes these approaches from DCS is the fact that fault tolerance properties are verified *a posteriori*. In contrast, DCS approaches *synthesize* automatically a controller that will insure the required fault tolerance properties by construction, that is *a priori*.

9 Conclusion and future work

9.1 Contribution

After introducing discrete controller synthesis (DCS) and its application to the automatic addition of fault tolerance in systems, we have presented in details how to specify and handle the failures of hardware components (processors, communication links, actuators, and sensors). Then we have shown how to specify and handle several kinds of failures (crash, value, and Byzantine). Finally, we have demonstrated with two case studies how our framework for fault tolerance can be used in practice. These case studies share the fact that the plant is specified as the synchronous product of several LTSs, with one LTS representing the fault hypothesis, and that the synthesis objective is specified as reachability and invariance predicates on states. Our research results are supported by a tool chain [1] (developed by us and by other research labs): MATOU to program LTSs in an easy way, and SIGALI for the DCS tool. The great advantages of our framework for fault tolerance are:

- It is **automatic**, because DCS produces automatically a fault tolerant system from an initial fault intolerant one.
- The **separation of concerns**, because the fault intolerant system can be designed independently from the fault tolerance requirements.
- The **flexibility**, because, once the system is entirely modeled, it is easy to try several fault hypotheses, several environment models, several fault tolerance goals, several degraded modes, and so on.
- The **safety**, because, in case of positive result obtained by DCS, the specified fault tolerance properties are guaranteed by construction on the controlled system.
- The **optimality** when optimal synthesis is used, modulo the potential numerical equalities (hence a non strict optimality).

If DCS fails w.r.t. the fault tolerance objective, then since *all* the state space is traversed during the synthesis (be it exhaustively or symbolically), it means that no solution exists for the required objective, fault hypothesis, environment model, and partition of the events into the controllable and the uncontrollable ones. The solution is then to relax one of these constrains, for instance to tolerate less failures.

[9] UPPAAL-TIGA: http://www.cs.aau.dk/~adavid/tiga

9.2 Discussion on complexity

The main drawback of our framework is the **combinatorial explosion**. This is a general drawback of DCS. Concretely, for large systems, the state space is too big to be traversed by a synthesis tool in a reasonable time. For some classes of problems, DCS can even be undecidable [54,52].

For the decidable part, our opinion is that DCS is today at the same level as model checking was 15 years ago, that is, it is a promising technique, but due to its algorithmic complexity it cannot be applied yet to industrial size systems. However, it must be noted that DCS does benefit from algorithmic and tools progress occurring in the model-checking area.

Furthermore, in our applicative setting, the problem of the algorithmic complexity can be tackled by defining appropriate methodologies. Our approach is to focus on the control kernel of a system, abstracting from the rest (e.g., numerical computations). Even though identifying the right level of description is more of a practical than a theoretical essence, it can have a vital impact on the concrete applicability of the techniques [1].

Finally, the synthesis of controllers is a constructive operation, so the complexity comparison should be made with the manual writing of controllers, followed by their verification and debugging. It is from that perspectives that we think that, in the case of the algorithms we use, their complexity remains reasonable, in the sense that they can be used for systems of a size where manual design would be very hard.

Regarding the results we have presented in this article, two points that can be improved w.r.t. scalability:

1. The DCS tool that we use, SIGALI, is very powerful thanks to its usage of a tri-valued logic ($\mathbb{Z}/3\mathbb{Z}$), but this comes at the price of less computational efficiency. This is embodied by two drawbacks: firstly the translation from Mode Automata (our language to specify LTSs) into $\mathbb{Z}/3\mathbb{Z}$ (the input format of SIGALI), and secondly the symbolic state space traversal by SIGALI, currently performed with TDDs, the ternary equivalent of BDDs, but alas less efficient. Nonetheless, DCS being a constructive method (in contrast with model-checking which is a diagnosis method[10]), we advocate that it is well worth spending some computation time to obtain **correct-by-construction fault tolerant systems**.
2. We would like to combine our results with abstract interpretation [20] to achieve the control of systems with both numerical and discrete data; this would allow us to pursue further our work on handling the value and Byzantine failures. Tools that implement efficiently abstract interpretation on LTSs exist, for instance NBAC [30].

9.3 Future work

In the framework we have presented so far, the result of the DCS is a centralized controlled system, fault tolerant provided that the synthesis objective includes a fault tolerance requirement (e.g., no task should be active on a faulty processor). However, it remains a centralized system, because it consists of a single global LTS, which is the result of the synchronous product of the plant and the synthesized controller. This can be a problem w.r.t. fault tolerance, since most fault tolerant systems must intrinsically be distributed to offer redundancy [25]. In particular, the controlled system should be tolerant to the failures of the controller.

The automatic generation of local controllers achieving global control objectives is a more difficult task, also known as **decentralized supervisory control** [38,18], among which we distinguish two cases: First the case where the local controllers do not communicate at run time, and second the case where the local controllers can exchange information at run time. However, there are two reasons that prevent us from using this technique. On the one hand, distributed DCS is not fault tolerant, since the failure of one local controller (e.g., following the failure of the processor it is running on) can lead to the failure of the whole system. And on the other hand, the distributed DCS problem without communication between local controllers has been shown to be undecidable [54,52].

Rather, we propose to distribute afterwards the controller. It can also be distributed manually when it is small enough, as demonstrated in [23]. The controller being an LTS, classical LTS distribution algorithm like [15] can be used. Without entering into the details, starting from a centralized LTS, this algorithm produces a set of communicating LTSs, one for each desired computing location, guaranteed to be semantically equivalent to the initial centralized LTS. Then, classical fault tolerance techniques can be used to make the communications between the local LTSs tolerant to the failures of the processors and the communication links.

Another track that we are considering currently involves addressing specifically software faults. Indeed, software faults could be addressed by modeling with behaviors such as n-version programming and voting mechanisms, and then by adapting the fault tolerant policy to this particular case.

[10] Some even say “autopsy”!

Finally, our framework and tool chain could be integrated within the NEMO compiler [21], which nicely integrates DCS as a compilation step of the domain-specific language for multi-task systems NEMO. This would provide a more integrated and easy to use fault tolerant framework.

Acknowledgments

Many thanks to Hervé Marchand for his expertise on discrete controller synthesis and on the SIGALI tool, to Karine Altisen for her work on the integration of SIGALI and Mode Automata (in particular the SIGALSIMU tool), to Emil Dumitrescu for his work on optimal discrete controller synthesis and his case study on the failures of communication links, to Huafeng Yu for his case study on water tanks conducted with synchronous observers, to Safouan Taha for his case study on water tanks conducted with conditioned synthesis objectives, and to Nour Brinis for taming the Byzantine generals.

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*, number 2618 in LNCS, Warsaw, Poland, April 2003.
2. K. Altisen, G. Gössler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Journal of Real-Time Systems*, 23(1/2):55–84, 2002.
3. P.C. Attie, A. Arora, and E.A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Trans. Programming Languages and Systems*, 26(1):125–185, 2004.
4. A. Avizienis, J.-C. Laprie, and B. Randell. Dependability and its threats: a taxonomy. In *IFIP World Computer Congress*, pages 91–120, Toulouse, France, August 2004. Kluwer Academic Pub., Hingham, MA.
5. R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
6. A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003. Special issue on embedded systems.
7. C. Bernardeschi, A. Fantechi, and L. Simoncini. Formally verifying fault tolerant system designs. *The Computer Journal*, 43(3), 2000.
8. B. Bonakdarpour and S.S. Kulkarni. Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In *International Conference on Distributed Computing Systems, ICDCS'07*, Toronto, Canada, June 2007.
9. B. Bonakdarpour and S.S. Kulkarni. Revising distributed UNITY programs is NP-complete. In *International Conference on Principles of Distributed Systems, OPODIS'08*, volume 5401 of *LNCS*, pages 408–427, Luxor, Egypt, December 2008. Springer-Verlag.
10. B. Bonakdarpour and S.S. Kulkarni. SYCRAFT: A tool for synthesizing distributed fault-tolerant programs. In *International Conference on Concurrency Theory, CONCUR'08*, volume 5201 of *LNCS*, pages 167–171, Toronto, Canada, August 2008. Springer-Verlag. Tool paper.
11. D. Brière, D. Ribot, D. Pilaud, and J.-L. Camus. Methods and specifications tools for Airbus on-board systems. In *Avionics Conference and Exhibition*, London, UK, December 1994. ERA Technology.
12. N. Brinis. Synthèse d'un contrôleur pour le problème des généraux byzantins. Master's Report, École Nationale des Sciences de l'Informatique, La Manouba, Tunisie, July 2005.
13. G. Bruns and I. Sutherland. Model checking and fault tolerance. In *International Conference on Algebraic Methodology and Software Technology, AMAST'97*, Sidney, Australia, 1997.
14. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8):677–691, 1986.
15. P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Trans. Software Engin.*, 25(3):416–427, May 1999.
16. F. Cassez, A. David, E. Fleury, K.G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *International Conference on Concurrency Theory, CONCUR'05*, volume 3653 of *LNCS*, pages 66–80, San Francisco (CA), USA, August 2005. Springer-Verlag.
17. K.-H. Cho and J.-T. Lim. Synthesis of fault-tolerant supervisor for automated manufacturing systems: A case study on photolothographic process. *IEEE Trans. Robotics and Automation*, 14(2):348–351, April 1998.
18. R. Cieslak, C. Desclaux, A. Fawaz, and P. Varaiya. Supervisory control of discrete-event processes with partial observations. *IEEE Trans. Automatic Control*, 33(3):249–260, March 1988.
19. J. Cortadella, A. Kondratyev, L. Lavagno, C. Passerone, and Y. Wanatabe. Quasi-static scheduling of independant tasks for reactive systems. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 24(10):1492–1514, October 2005.
20. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Symposium on Principles of Programming Languages*, Los Angeles (CA), USA, January 1977. ACM SIGPLAN.

21. G. Delaval and E. Rutten. A domain-specific language for multi-task systems, applying discrete controller synthesis. *EURASIP J. on Embedded Systems*, 2007. Article ID 84192.
22. E. Dumitrescu, A. Girault, H. Marchand, and E. Rutten. Optimal discrete controller synthesis for modeling fault-tolerant distributed systems. In *Workshop on Dependable Control of Discrete Systems, DCDS'07*, pages 23–28, Cachan, France, June 2007. IFAC, New-York.
23. E. Dumitrescu, A. Girault, and E. Rutten. Validating fault-tolerant behaviors of synchronous system specifications by discrete controller synthesis. In *Workshop on Discrete Event Systems, WODES'04*, Reims, France, September 2004. IFAC, New-York.
24. E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.
25. F. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, March 1999.
26. F. Gärtner and A. Jhumka. Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications. In *Joint Conference on Formal Modelling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault Tolerant System, FORMATS-FTRTFT'04*, volume 3253 of *LNCS*, Grenoble, France, September 2004. Springer-Verlag.
27. A. Girault and E. Rutten. Discrete controller synthesis for fault-tolerant distributed systems. In *International Workshop on Formal Methods for Industrial Critical Systems, FMICS'04*, volume 133 of *ENTCS*, pages 81–100, Linz, Austria, September 2004. Elsevier Science, New-York.
28. A. Girault and H. Yu. A flexible method to tolerate value sensor failures. In *International Conference on Emerging Technologies and Factory Automation, ETFA'06*, pages 86–93, Prague, Czech Republic, September 2006. IEEE, Los Alamitos, CA.
29. N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *International Conference on Algebraic Methodology and Software Technology, AMAST'93*, Twente, NL, June 1993. Springer-Verlag.
30. B. Jeannet. Dynamic partitioning in linear relation analysis. Application to the verification of reactive systems. *Formal Methods in System Design*, 23(1):5–37, July 2003.
31. R.M. Jensen, M. Veloso, and R. Bryant. Synthesis of fault-tolerant plans for non-deterministic domains. In *Workshop on Planning under Uncertainty and Incomplete Information*, Trento, Italy, June 2003.
32. O. Kamach, L. Pietrac, and E. Niel. Approche multi-modèle pour les systèmes à événements discrets: application à un préhenseur pneumatique. In *Modélisation des Systèmes Réactifs, MSR'05*, pages 159–174, Autrans, France, September 2005. Hermes.
33. S.S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In M. Joseph, editor, *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT'00*, volume 1926 of *LNCS*, pages 82–93, Pune, India, September 2000. Springer-Verlag.
34. S.S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. In *International Conference on Dependable Systems and Networks, DSN'04*, Firenze, Italy, June 2004. IEEE, Los Alamitos, CA.
35. S.S. Kulkarni and A. Ebnenasir. Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Trans. Dependable Secure Comput.*, 2(3):201–215, July 2005.
36. R. Kumar and V.K. Garg. Optimal supervisory control of discrete event dynamic systems. *SIAM J. Control Optim.*, 33(2):419–439, 1995.
37. L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Programming Languages and Systems*, 4(3):382–401, July 1982.
38. F. Lin and W.M. Wonham. Decentralized supervisory control of discrete-event systems. *Information Sciences*, 44(3):199–224, April 1988.
39. F. Lin and W.M. Wonham. On observability of discrete-event systems. *Information Sciences*, 44(3):173–198, April 1988.
40. 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.
41. H. Marchand, O. Boivineau, and S. Lafortune. On the synthesis of optimal schedulers in discrete event control problems with multiple goals. *SIAM J. Control Optim.*, 39(2):512–532, 2000.
42. H. Marchand, O. Boivineau, and S. Lafortune. On optimal control of a class of partially observed discrete event systems. *Automatica*, 38:1935–1943, 2002.
43. 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.
44. 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.
45. H. Marchand and M. Samaan. Incremental design of a power transformer station controller using a controller synthesis methodology. *IEEE Trans. Software Engin.*, 26(8):729–741, August 2000.
46. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall, 1989.
47. D. Powell. Failure mode assumption and assumption coverage. In *International Symposium on Fault-Tolerant Computing, FTCS-22*, pages 386–395, Boston (MA), USA, July 1992. IEEE, Los Alamitos, CA. Research report LAAS 91462.
48. P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, January 1987.

49. H. Schepers and J. Hooman. Trace-based compositional proof theory for fault tolerant distributed systems. *Theoretical Computer Science*, 128, 1994.
50. R. Sengupta and S. Lafortune. An optimal control theory for discrete event systems. *SIAM J. Control Optim.*, 36(2):488–541, March 1998.
51. S. Taha. Synthèse de contrôleurs discrets pour systèmes embarqués tolérants aux pannes. Master's Report, Institut National Polytechnique de Grenoble, Grenoble, France, June 2004.
52. S. Tripakis. Decentralized control of discrete event systems with bounded or unbounded delay communication. *IEEE Trans. Automatic Control*, 49(9):1489–1501, September 2004.
53. E. Tronci. Optimal finite state supervisory control. In *IEEE Conference on Decision and Control, CDC'96*, Kobe, Japan, December 1996. IEEE, Los Alamitos, CA.
54. J.N. Tsitsiklis. On the control of discrete event dynamical systems. *Mathematics of Control, Signals, and Systems*, 2(2):95–107, June 1989.