

A flexible method to tolerate value sensor failures

Alain Girault*

Huafeng Yu†

Abstract

Tolerating the value failures of sensors is an important problem in automated control processes and plants. In this paper, we address this problem in a theoretical framework in order to demonstrate the feasibility of an automatic method based on discrete controller synthesis. We consider a fault-intolerant program whose job is to control an automated process, here a liquid tank equipped with level sensors that can be subject to value faults. This fault-intolerant program is modeled as a finite labeled transition system. We then specify formally a fault hypothesis, i.e., how many sensors can fail simultaneously. We use discrete controller synthesis to obtain automatically a program, having the same behavior as the initial fault-intolerant one, and satisfying the fault tolerance requirements under the fault hypothesis. We advocate that, thanks to the use of discrete controller synthesis, our method offers flexibility, reliability, separation of concern, and it is automatic.

Keywords. Sensors with value failures, discrete controller synthesis, automatic fault tolerance.

1 Introduction to discrete controller synthesis

Discrete controller synthesis (DCS) was invented by Ramadge and Wonham in the nineteen eighties [19]. 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:

$$\mathcal{U} \cap \mathcal{C} \subseteq \mathcal{D} \quad (1)$$

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 possible a behavior of the plant (i.e., a sequence of events). Ramadge and Won-

ham have implemented a tool suite for DCS, called TCT.¹

Since \mathcal{U} and \mathcal{D} are given and we want to find \mathcal{C} such that $\mathcal{U} \cap \mathcal{C} \subseteq \mathcal{D}$, in some sense the solution could be written as $\mathcal{C} = \mathcal{D}\mathcal{U}^{-1}$, provided that we define the operators ‘.’ and ‘⁻¹’ on languages. Hence, 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 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 $\langle \text{plant}, \text{controller} \rangle$ 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.

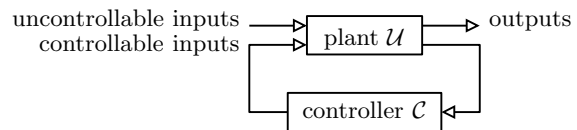


Figure 1: The closed loop system $\langle \mathcal{U}, \mathcal{C} \rangle$.

More recently, several research teams have applied and extended these language theory techniques to **labeled transition systems** (LTS), both in the area of computer science and automatic discrete control. 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 SIGALI² [18], tool that we have used in the present article.

2 Motivation

In automated process control, fault-tolerance is a crucial issue that must be addressed when designing the system [1]. Moreover, value faults are an important class of faults for this kind of systems.

*INRIA Rhône-Alpes, POP ART project, 655 avenue de l'Europe, 38334 Saint-Ismier cedex, France, Email: Alain.Girault@inrialpes.fr

†INRIA Futurs / LIFL, DART project, USTL bât. M3, 59655 Villeneuve d'Ascq cedex, France, Email: Huafeng.Yu@lifl.fr

¹<http://www.control.utoronto.ca/people/profs/wonham>.

²<http://www.irisa.fr/vertecs/Logiciels/sigali.html>.

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, 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 5.

In this article, we propose a DCS-based method to transform automatically a fault-intolerant program into a fault-tolerant one. It offers the following advantages:

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

We start by introducing the formal model of labeled transition systems, how they are used in DCS, and the general principles for automating the addition of fault tolerance with DCS: this is Section 3. Then, we present in Section 4 our model of a liquid tank equipped with several sensors that can be affected by value failures; we present in details how DCS can be used to produce automatically a controller that will make the system tolerant to such value failures. We end with a presentation of the related work in Section 5, and with some concluding remarks in Section 6.

3 Formal models used in DCS

3.1 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 an input alphabet (i.e., a finite set of input events produced by the environment), \mathcal{O} is an output alphabet (i.e., a finite set of output events emitted towards the environment), and \mathcal{T} is the transition relation, that is a subset of $\mathcal{Q} \times \mathcal{Bool}(\mathcal{I}) \times \mathcal{O}^* \times \mathcal{Q}$, where $\mathcal{Bool}(\mathcal{I})$ is the set of boolean expressions of \mathcal{I} .

Each transition has a **label** of the form g/a , where $g \in \mathcal{Bool}(\mathcal{I})$ 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.

The composition operator of two LTSs put in parallel is Milner's **synchronous product**, noted \parallel . The synchronous product is commutative and associative. Formally:

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

Like all product operators for LTSs, the synchronous product causes a *combinatorial explosion*, since the number of states in $S_1 \parallel S_2$ 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, thanks to the putting in common of successor states.

A **path** in the LTS $\langle \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} \rangle$ is a sequence of transitions $q_1 \xrightarrow{g_1 / a_1} q_2 \xrightarrow{g_2 / a_2} q_3 \dots q_n \xrightarrow{g_n / a_n} q_{n+1}$. A state q of \mathcal{Q} is **reachable** iff there exists a path from the initial state q_0 to q . A set of states E is **reachable** iff all its states are reachable. A set of states E is **invariant** iff any transition having as source a state of E has its destination state in E .

We use the Mode Automata language to program LTSs [17]. Without entering in the 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, which is the input format of the SIGALI tool for DCS [18].

3.2 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. 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. 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**.

\mathcal{D} is the objective that the controlled system must fulfill, typically *to make invariant a subset of \mathcal{U}* , or *to keep unreachable a subset of \mathcal{U}* . In model checking, it is well known that such objectives can be equivalently expressed as predicates on the states of \mathcal{U} or as LTSs.

The controller \mathcal{C} obtained with DCS achieves this objective by *restraining* the transitions of \mathcal{U} , that is by *cutting* those that would jeopardize the objective \mathcal{D} . \mathcal{C} can only cut the controllable transitions. To do this, it chooses a valuation of the controllable inputs such that the guard of the transition is false. It is the most permissive controller: if, in a given state, it has the choice between two controllable transitions that both satisfy \mathcal{D} , then the controller does not choose a priori between them, and thus the resulting controlled system is *non deterministic*. But of course, one transition can be chosen arbitrarily at run time to make the controlled system *deterministic*.

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 cutting controllable transitions. This can be solved by relaxing the control objective \mathcal{D} or by adding more controllability to the plant \mathcal{U} .

3.3 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. Thus, on one hand 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 one fault prevents the system from providing its nominal service). On the other hand, 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 prop-

erties of \mathcal{D} , we will obtain *automatically* a fault-tolerant system.

Besides, the fault hypothesis will be modeled 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 since it is possible to change the fault hypothesis without modifying the remaining of the specification (separation of concerns), and second it is formal thanks to the usage of an LTS.

4 A liquid tank with sensors

DCS being limited to Booleans, it has been necessary to restrict ourselves to sensors returning a true/false value. We have therefore chosen to study a **liquid tank** equipped with n level sensors, subject to value failures, and with **three reliable valves**. The purpose of this tank is to store some liquid and to be used by somebody, for instance by another process. Hence, the goal is to control the valves in such a way that the tank be neither empty nor over-flooding. Other control objectives can be specified as well, as we will show in the sequel. Figure 2(c) illustrates our liquid tank.

We note C_1, C_2, \dots, C_n the sensors. Each can be either wet or dry, which can indeed be represented by a true/false value. The fault hypothesis is that the n sensors have permanent value faults, and that *some* of them can be faulty. Each sensor C_i can therefore be modeled by the LTS of Figure 2(a): M_i is the wet state, S_i is the dry state, and ERR_i is the error state. The fact that faults are permanent is modeled by the absence of transition from the ERR_i state to the S_i or M_i states. According to the initial liquid level in the tank, the initial state of the LTS is either S_i or M_i . For instance, if the initial level is anywhere between the sensors C_i and C_{i+1} , then the initial states of the sensors will be $M_1, \dots, M_i, S_{i+1}, \dots, S_n$. Moreover, the LTS of sensor C_i has a Boolean output, noted c_i , equal to false if the sensor is dry, and true if it is wet. For instance, the transition $S_i \xrightarrow{\bar{f}_i m_i / 1} M_i$ models the

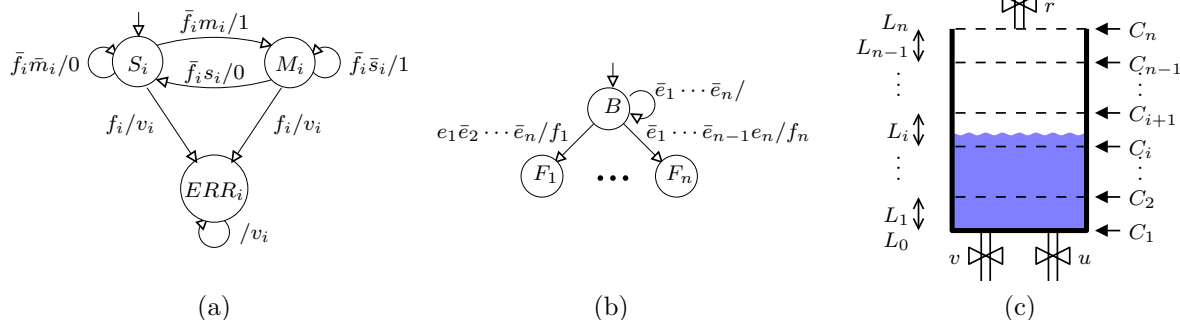


Figure 2: (a) LTS of sensor C_i and (b) of the fault hypothesis; (c) Tank with n sensors and 3 valves.

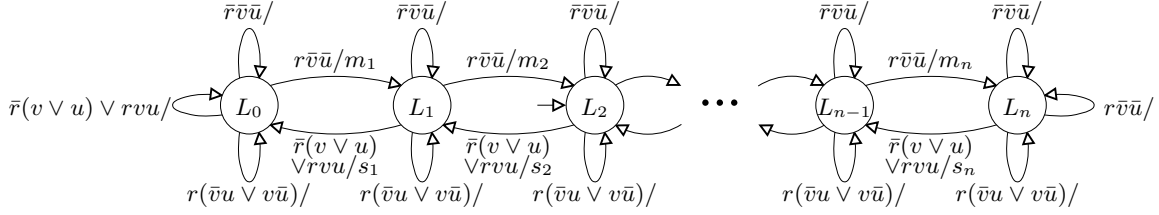


Figure 3: LTS modeling the tank of Figure 2(c).

fact that the sensor C_i becomes wet and outputs $c_i = 1$. Finally, v_i is an uncontrollable Boolean input that models the fact that, when the sensor C_i is faulty, its output can take any value. For instance, the transition $S_i \xrightarrow{f_i/v_i} ERR_i$ models the fact that the sensor C_i becomes faulty and outputs $c_i = v_i$.

The fault hypothesis must state how many of the n sensors can become faulty. Again, we formally model this with a LTS. For instance, the LTS of Figure 2(b) models the hypothesis that at most one sensor can become faulty. This LTS has one “level” of faulty states, named F_i , with the convention that in F_i , only the sensor C_i is faulty. If two sensors can become faulty, then it will have two levels of faulty states, where the states in the first level will represent the failure of one sensor, and the states in the second level will represent the cumulative failures of two sensors. Like in the previous section, the events e_i are uncontrollable while the events f_i are local. To avoid too complex drawings, the self-transitions around the F_i states, which make the LTS reactive, have been omitted in Figure 2(b).

When all the sensors are dry, the tank is empty: this level is noted L_0 . In contrast, when all the sensors are wet, the tank is over-flooding: this level is noted L_n . In between, when sensors C_1 to C_i are wet and sensors C_{i+1} to C_n are dry, the level is noted L_i . The levels L_1, L_2, \dots, L_{n-1} are therefore *abstractions*, since actually the exact level of the liquid in the tank can be anywhere between the sensors C_i and C_{i+1} .

Furthermore, the tank is equipped with three valves: one to fill the tank, noted r (controllable), one to empty the tank, noted v (controllable), and one for the user to tap liquid from the tank, noted u (uncontrollable). The event r indicates that the filling valve is open, while the event \bar{r} indicates that is closed (and similarly for the two other valves). For the sake of simplicity, we assume that all the valves have exactly the same flow per time unit.

The LTS that models the liquid tank is shown in Figure 3. It has $n + 1$ states, one for each abstract level of the liquid. Its inputs are the events r , v , and u , while its outputs are the events m_i and s_i , $1 \leq i \leq n$, to trigger, in the LTSs of the sensors, the state changes corresponding to the liquid level

change. If the flow per time units is not identical for each valve, then this LTS must be modified accordingly.

Figure 4 shows an example of a complete system made of a tank, n sensors, and the fault hypothesis. In the present case, the LTS \mathcal{U} of the plant is the result of the synchronous product of the LTS of the tank, the n LTSs of the sensors, and the LTS of the fault hypothesis.

The goal of DCS is to control the valves in such a way that the tank be neither empty nor over-flooding. We are therefore looking for a controller that will guarantee that the states L_0 and L_n of the LTS of Figure 3 are unreachable. This is formally expressed by the following property:

$$\neg(L_0 \vee L_n) \quad (2)$$

Other synthesis objectives can be expressed as well. For instance, if we want to guarantee that the level will always remain between the sensors C_i and C_j (with $i < j$), then the objective becomes:

$$\text{Invariant}(\{L_i, \dots, L_{j-1}\}) \quad (3)$$

where the *Invariant* (X) predicate checks if the set of states X is invariant.

Since the property (2) is on the state variables of the tank (L_0 and L_n), it means that the synthesized controller interacts with the tank by observing its state and by acting upon the controllable variables r and v , and not by observing the outputs of the n sensors. This situation is illustrated in Figure 5. So, on the one hand DCS guarantees by construction that L_0 and L_n are unreachable in the controlled system (which is what we want), but on the other the controller does not observe the outputs of the sensors, so it produces the events r and v without taking into account the possible faults that can affect the sensors (which is not what we want).

To solve this problem, we propose to add a **synchronous observer** to the system, exactly like in model checking [10] (and unlike Wong and Wonham’s observers that are causal reporter maps with the observer property [22]): it is an LTS whose job is to observe the outputs c_i of the sensors, and to go in state *BAD* as soon as these outputs correspond to the empty tank (that is, when all the sensors are

dry: $\bigwedge_{i=1}^n \bar{c}_i$) or to the over-flooding tank (that is, when all the sensors are wet: $\bigwedge_{i=1}^n c_i$). Thanks to this observer, our DCS objective becomes simply: $\neg BAD$. This time, the controller acts upon the inputs r and v according to the outputs of the sensors, and thus according to their possible faults. Its LTS is shown in Figure 6(a), and the new controlled system is shown in Figure 6(b).

Once the new controller is obtained with DCS, there remains to prove that the states L_0 and L_n are indeed unreachable in the controlled system, since this property is not anymore guaranteed by the objective of the DCS (because the synthesis objective is now on the BAD state and not anymore on the L_i states). Such a formal proof can be achieved with a model checking tool, by checking the following property on the controlled system $\mathcal{U}||\mathcal{C}$:

$$\neg \text{Reachable}(\{L_0, L_n\}) \quad (4)$$

where the $\text{Reachable}(X)$ predicate checks if the set of states X is reachable.

We have programmed this liquid tank with MATOU and SIGALI, in the particular case $n = 4$ (i.e., four sensors). MATOU compiles an LTS into a set of polynomial equations in $\mathbb{Z}/3\mathbb{Z}$, which is the input format of SIGALI ($\mathbb{Z}/3\mathbb{Z}$ being the Galois field with three elements, $\{-1, 0, 1\}$). SIGALI is used both for synthesizing the controller and for model checking the resulting controlled system. The uncontrolled system \mathcal{U} contains 813 states and the controlled system 243, for a total of 47 state variables; SIGALI allows us to isolate the polynomial equation in $\mathbb{Z}/3\mathbb{Z}$ that corresponds to the controller alone \mathcal{C} , and to transform it into a TDD (“Ternary Decision Diagram”, ternary equivalent of BDDs); in the present case, the size of ASCII representation of the obtained TDD is 5.9 MB.

We have been able to formally check with SIGALI that the empty and the over-flooding states (L_1

and L_4 respectively) are unreachable in the resulting controlled system $\mathcal{U}||\mathcal{C}$. This proves that the controlled system is tolerant to one sensor’s value fault. However, if we modify the fault hypothesis to try to tolerate two sensor faults, then DCS fails; this means that, in the case of value faults, two faulty sensors among four make the system uncontrollable. In other words, our liquid tank equipped with four sensors can tolerate exactly one value fault. To the best of our knowledge, this was the first time DCS was used to tolerate value faults.

The proposed method is quite general, and can be adapted to other types of failures. For instance, in previous work we have successfully used it to tolerate processor failures in the context of a multi-processor multi-task real time system [9], communication link failures in the context of a distributed bus arbiter [7], and actuator failures in the context of a double tank of liquid connected by pipes [21]. The common point shared by these four usages of DCS (including the one presented in this paper) is that the failures are modeled by uncontrollable events, and that DCS allows us, first to prove that a given system does tolerate a specified number of failures, and second to obtain automatically a controller that, combined with the initial fault-intolerant system, makes the resulting system fault-tolerant.

5 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 [5]. Their results are based on the framework of supervisory control of discrete event systems of Ramadge and Wonham [19]. First, the set of events Σ is partitioned into $\Sigma = \Sigma_c \cup \Sigma_{uc} = \Sigma_n \cup \Sigma_{an}$, respectively

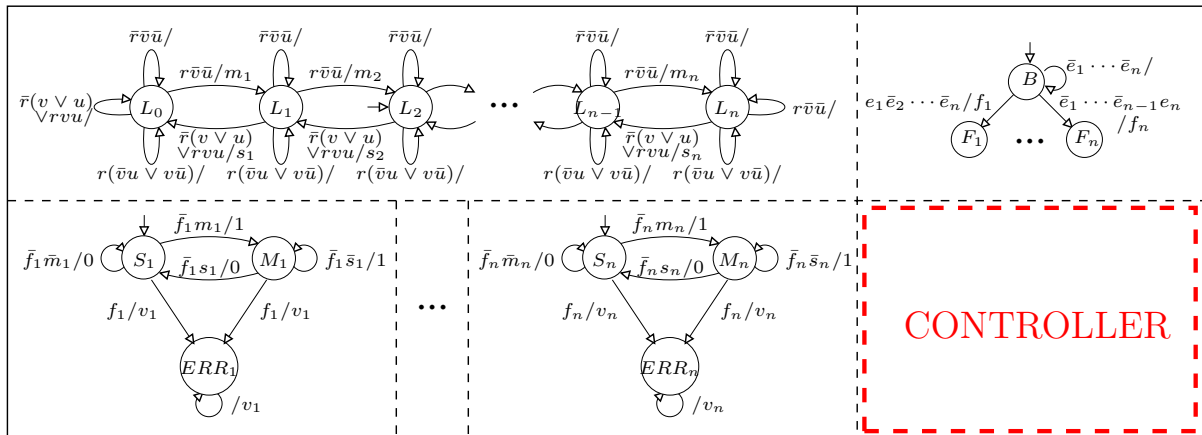


Figure 4: Complete system made of the tank, n sensors, and the fault hypothesis.

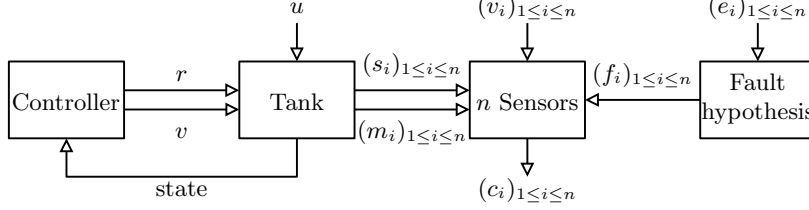


Figure 5: Controller interacting with the plant as in an open loop system.

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 is:

- 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;
- finally, our usage of a DCS software tool while Cho and Lim do not mention such a tool.

Although they never mention DCS, the technique proposed by Kulkarni and Arora in [15], and improved by Kulkarni and Ebneenasir in [16], is very 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 states, each state being a valuation of the program's variables, and a set of transitions. 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't. 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.

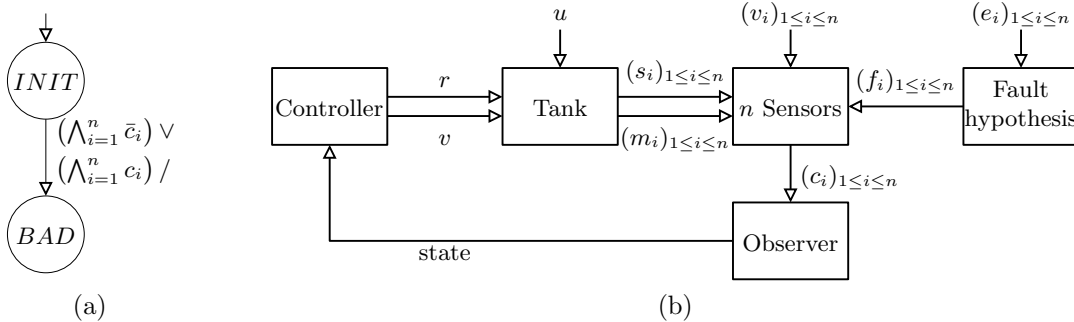


Figure 6: (a) Synchronous observer for the tank and sensors system; (b) Closed loop system.

To address their two models of atomicity and their three levels of fault tolerance, the authors propose a sound algorithm that is polynomial (resp. exponential) in the state space of the source fault-intolerant program. In the low atomicity model, the transformation problem is NP-complete. Each transformation involves recursively removing bad transitions. This principle of program transformation implies that the initial fault-intolerant program should be maximal (weakest invariant and maximal non-determinism). As we can see, such an automatic program transformation is very similar to DCS.

Based on the work of Kulkarni and Arora, Gärtner and Jhumka propose a way to deal also with non fusion closed traces [8]. 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 fusion closed specification is $\mathcal{O}(n^2)$.

Kamach et al. have applied DCS to a system with several modes of operations [14]. 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.

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. [13]. 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 BBDs [4]. The main limitation of their results is that they tolerate only one fault, while our model can accommodate an

arbitrary number of faults.

Formal approaches to the design of fault-tolerant systems have mostly considered the problem of **formal verification**, in the context of process algebra [20, 3, 2]. 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* [12].

6 Conclusion

After introducing discrete controller synthesis (DCS) and its application to the automatic addition of fault tolerance in systems, we have presented a novel approach to produce automatically systems tolerant to value faults.

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. The great advantages of our approach 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, and so on.
- The **reliability**, because, in case of positive result obtained by DCS, the specified fault tolerance properties are guaranteed by construction on the controlled system.

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 provided 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 constraints, for instance to tolerate less failures.

The main drawback 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. 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 of a limited usage due to its algorithmic complexity. Regarding the results we have presented in this article, two points can be improved w.r.t. scalability; on the one hand the translation from Mode Automata (our language to specify LTSs) into $\mathbb{Z}/3\mathbb{Z}$ (the input format of SIGALI) and on the other the symbolic state space traversal by SIGALI, currently performed with TDDs, the ternary equivalent of BDDs, but alas less efficient. Also, we would like to combine our results with abstract interpretation [6], in order to treat numerical values systems, that is systems having value faults. Tools that implement efficiently abstract interpretation on LTSs exist, for instance NBAC [11].

References

- [1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1):11–33, January 2004.
- [2] C. Bernardeschi, A. Fantechi, and L. Simoncini. Formally verifying fault tolerant system designs. *The Computer Journal*, 43(3), 2000.
- [3] G. Bruns and I. Sutherland. Model checking and fault tolerance. In *AMAST'97*, Sidney, Australia, 1997.
- [4] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, 1986.
- [5] K.-H. Cho and J.-T. Lim. Synthesis of fault-tolerant supervisor for automated manufacturing systems: A case study on photolithographic process. *IEEE Trans. on Robotics and Automation*, 14(2):348–351, April 1998.
- [6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, Los Angeles, USA, January 1977.
- [7] E. Dumitrescu, A. Girault, and E. Rutten. Validating fault-tolerant behaviors of synchronous system specifications by discrete controller synthesis. In *WODES'04*, Reims, France, September 2004.
- [8] F. Gärtner and A. Jhumka. Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications. In *FORMATS-FTRTFT'04*, volume 3253 of *LNCS*, Grenoble, France, September 2004. Springer-Verlag.
- [9] A. Girault and E. Rutten. Discrete controller synthesis for fault-tolerant distributed systems. In *FMICS'04*, ENTCS, Linz, Austria, September 2004. Eslevier Science.
- [10] 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, *AMAST'93*, Twente, NL, June 1993. Springer-Verlag.
- [11] 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.
- [12] R. Jensen. DES controller synthesis and fault tolerant control – a survey of recent advances. Research report TR-2003-40, ITU, Copenhagen, Denmark, December 2003.
- [13] R. 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.
- [14] O. Kamach, L. Pietrac, and E. Niel. Approche multi-modèle pour les systèmes à événements discrets: application à un préhenseur pneumatique. In *MSR'05*, pages 159–174, Autrans, France, September 2005. Hermes.
- [15] S.S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In M. Joseph, editor, *FTRTFT'00*, volume 1926 of *LNCS*, pages 82–93, Pune, India, September 2000. Springer-Verlag.
- [16] S.S. Kulkarni and A. Ebnesnasir. Automated synthesis of multitolerance. In *DSN'04*, Firenze, Italy, June 2004. IEEE.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] H. Schepers and J. Hooman. Trace-based compositional proof theory for fault tolerant distributed systems. *Theoretical Computer Science*, 128, 1994.
- [21] S. Taha. Synthèse de contrôleurs discrets pour systèmes embarqués tolérants aux pannes. Master's Report, Institut National Polytechnique de Grenoble, Grenoble, France, June 2004.
- [22] K.C. Wong and M.W. Wonham. On the computation of observers in discrete-event systems. *Discrete Event Dynamic System: Theory and Applications*, 14(1):55–107, January 2004.