A self_updating model for analysing system reconfigurability

Anne-Lise Gehin, Hexuan Hu, Mireille Bayart

A self-updating model for analysing system reconfigurability

Anne-Lise Gehin[a], Hexuan Hu[b], Mireille Bayart[a]

[a] *LAGIS FRE CNRS 3303, Université Lille 1, 59655 Villeneuve d'Ascq cedex, France*
[b] *Hohai University Nanjing, Jiangsu Province, China*

Abstract

Systems are built by connecting different components (e.g., sensors, actuators, process components) that are, in turn, organized to achieve system objectives. But, when a system component fails, the system's objectives can no longer be achieved. For many years, numerous studies have proposed efficient fault detection and isolation (FDI) and fault-tolerant control (FTC) algorithms. This paper considers faults that lead to the complete failure of actuators. In this specific case, the system's physical structure changes, and the system model thus becomes incorrect. The potential that the system has to continue to achieve its objectives has to be re-evaluated from a qualitative point of view, before recalculating or modifying the control algorithms. To this end, this paper proposes a self-updating system model to reflect the current system potential, a formulation of system objectives using temporal logic, and a verification method based on model checking to verify if the objectives can still be achieved by the faulty system. The systems considered are discrete-continuous systems.

Keywords: self-updating model, reconfigurability analysis, fault-tolerant control, model checking

1. Introduction

Control systems are expected to achieve different objectives at different times (e.g., production objectives, quality objectives, safety objectives). The achievement of these objectives relies on the services provided by the system components (e.g., sensors, actuators, process components). If a component fails, the system objectives can be compromised [1]. For this reason, in order to meet the increasing demand for system safety and reliability, control systems are increasingly integrating fault detection and isolation (FDI) and fault-tolerant control (FTC) procedures [2].

FDI algorithms aim to detect and the localize the faults [1] as quickly as possible, in order to make decisions that will avoid their propagation and their

[1] The literature gives slightly different meanings for the notions of "fault" and "failure". Isermann provides a survey related to the standardisation of these two notions in chapter 2.2

undesirable effects. They check the consistency of the real time system operation observations using the available model-based or data-based knowledge. Venkatasubramanian et al. has published a good review of the FDI methods, classified in quantitative, qualitative or process history categories [4], [5], [6]. FTC concerns the potential for the system to continue its operations with the required performances, despite component failures. FDI algorithms, being part of the information system of the supervised process, need information redundancy, whereas FTC algorithms, being part of the decision and actuation system, need decision and actuation redundancy, (i.e., the redundancy of the services that are provided to the users by the system components) [7].

Different approaches have been proposed for the design of FTC algorithms [8], [9]. In passive approaches, the controllers are fixed and designed to be robust against a class of presumed faults. In passive FTC, a system may tolerate only a limited number of faults, which are assumed to be known before designing of the controller. The fault considered are time-varying processes or uncertain parameters in the system component models. Suppose, for example, that a pump is used to regulate the level in a tank and that a regulator is used to produce the pump control $v(t)$ so as to obtain the pump flow $Q = k_v.v(t)$, where $k(v)$ is uncertain (i.e., not well identified) or is time varying because there is a leak that depends on the pump's rotational speed; thus, the delivered flow Q will not have the desired value. In passive approaches, the controller parameters are adapted from an estimation of the value $k(v)$ (adaptive control) or from a minimization of a predictive criterion (robust control). (Readers interested in passive FTC approaches can referred to references [10] and [11] for a good introduction. Readers interested in adaptive or robust control methods can refer to references [12], [13].)

Compared to the passive approaches, an active FTC system uses the available resources and both physical and analytic system redundancy to deal with unanticipated faults, either by selecting a pre-computed law, or by synthesizing a new one in real time [14], [15]. After a failure, a part of the services provided by the system's components may become unavailable and the overall system will work inadequately, if, for example, a faulty sensor (resp. actuator) makes the plant partially unobservable (resp. uncontrollable). Objectives can be achieved, in spite of the faults, if at least one of two procedures - fault accommodation and system reconfiguration - can be used successfully. Fault accommodation relies on estimating the variables provided by the faulty sensors; system reconfiguration uses alternative actuators or sensors to provide services that are equivalent to the services previously provided by the failed components. Systems reconfiguration includes control reconfiguration: designing or the selecting a new law to control the new components implied in the new control loops [16].

---

of his book [3]: a fault is defined as an un-permitted deviation of at least one characteristic system property (i.e., feature) from the acceptable, usual or standard condition and a failure is a permanent interruption of a system's ability to perform a required function under specified operation conditions. Since a failure results from one or more faults and since a fault may initiate a failure or a malfunction, these two terms are used indifferently in this paper.

As this paper deals with actuator faults, and as sensors are assumed to be non-failing, only system reconfiguration will be considered. More precisely, what is meant by reconfiguration in this paper is, despite faulty components, the possibility of:

1. continuing system operation without intolerable loss of performance,
2. continuing system operation with reduced specifications, or
3. abandoning the mission while still avoiding disaster.

Different approaches have been proposed to the design of FTC algorithms under actuator faults [17], [18], [19]. But, in most of cases, these approaches use the quantitative system behaviour model including, for example, the state and output equations in the time domain to select or redesign the input vector, the output vector, the control law and the set-point. But in the case of a complete failure of an actuator, the potential for the system to achieve its objectives has to be re-evaluated qualitatively before synthesizing of a new control law. To express these possibilities in term of available or unavailable functions is a very precious understandable information for operators who supervise the system and who have to exactly know at each time what they can expected for this system.

In this paper, we propose analysing system reconfigurability using a self-updating model and a model-checking technique. The idea is to provide the mechanisms to automatically calculate the real system model each time a fault occurs and then to evaluate whether or not this updated model will be able to achieve the system's objectives. In computer science logic, model checking refers to a method for formally verifying that a model meets a given specification [20], [21], [22]. System specifications are expressed as temporal logic formulas, and models take the form of state transition graphs. Efficient symbolic algorithms are used to test whether or not a given model satisfies the given specifications; if not, the algorithm generates a counterexample. By studying the counterexample, the source of the error in the model can be highlighted and the model corrected. Model checking has proven to be a successful technique to verify the requirements and design of a variety of real-time embedded, safety-critical systems [23], [24]. Its application to analysing the reconfigurability of a system, is, we think, an original idea.

Our reconfigurable control architecture is presented in Figure 1. Two parts can be distinguished: diagnosis and reconfiguration. The diagnosis part aims to detect and identify faults. A fault is detected when the observations made on the real system are inconsistent with the prediction of the system model. A fault identification module is then run to identify the faulty component and its fault type (e.g., valve blocked in the open position or in the closed position for a valve). In the opposite case, the behaviour of the system is considered as normal (non-faulty), and the current system model remains valid. The reconfiguration part of our reconfigurable control architecture uses the result of the diagnosis part to automatically calculate an updated model, using the real observations and the fault reports. If the faulty system, as it is described by its updated model, allows the system to achieve its specified objectives, the updated model becomes

Figure 1: The reconfigurable control architecture.

the new current system model. In the opposite case, the system reconfiguration is considered as impossible, and the system is shut down or placed to a safe state.

This paper focuses on the reconfiguration part, so the diagnosis part is not described. The diagnosis is assumed to be correctly performed by an appropriate procedure. The systems considered are continuous systems for which continuous variables are discretised, and the faults are assumed to be permanent, not intermittent. The rest of the paper is organized as follows. Section 2 describes a system from a functional viewpoint. Section 3 presents our framework for generating a self-updating system model. Section 4 introduces the temporal logic used for describing system objectives and the model checking method used to automatically calculate reconfigurability. Section 5 presents the example used to illustrate the different notions. Section 6 provides our conclusions and some prospects for future research.

2. Functional viewpoint

Systems are built by connecting different components, which provide services to users [7]. For example, a sensor provides a measurement service; a pump provides two services: delivering a flow and not delivering a flow; a controller provides a calculation service; and a tank provides a storage service. Lower level components (e.g., sensors, actuators, controllers and process components) are grouped to form subsystems in order to define higher level services. For example, a level sensor, a pump, a controller and a tank can form a single subsystem that provides a regulation service in which the controller uses the tank level provided by the measurement service to calculate the pump's input control. The advantage of creating high-level components is to provide high-level services, which ends up in the overall system and its control objectives.

A service is described by the variables it consumes *(cons)*, the variables it produces *(prod)*, and a procedure *(proc)* that transforms the former into the latter. Services are derived from the component behaviour, which is governed by physical laws and (possibly) by embedded software. For example, a tank consumes input and puts out mass flows, and produces a stored mass, using the procedure $\dot{m}(t) = q_{in}(t) - q_{out}(t)$, where m is the stored mass, q_{in} is the flow in the input pipe, and q_{out} is the flow in the output pipe. The procedure follows from the principle of conservation of mass. The *regulation* service of a controller consumes data provided by a measurement service and produces signals to an actuation service according to a specific algorithm.

Services describe what the user expects to obtain from a component or from a subsystem under normal operation. However, there are two reasons for a given service s to fail to deliver the appropriate values of the variables it produces:

- *Internal faults* affect some resources *(res)* needed by the service. As a result, the actual values of the produced variables *(prod)* are not those specified by the procedure *(proc)*. A leak in a tank is an example of an internal fault. The procedure $\dot{m}(t) = q_{in}(t) - q_{out}(t)$ does not correctly describe the behaviour of a leaking tank since the flow associated with the leak is not taken into account.

- *External faults* affect the inputs *(cons)* of the service. A level regulation service is subject to an external fault when the level value it consumes is false, due to the failure of the level sensor, or if its time stamp is outdated, due to a failure in the communication system.

Fault-tolerant components integrate multiple instances of the same service, listed as a set of versions. All versions of the same service produce the same outputs (so they can be interchanged), and at least one among the inputs, procedures and resources is different from one version to another. The different versions of this same service differ by their accuracy, running time and/or energy consumption, and they are ranked by the designer. When a service is requested, the version that is selected is the most preferred in that all the resources it needs are known to be non-faulty. The first version of this ordered set is named *nominal version*, and the other versions are *degraded versions*. The service becomes unavailable if there is not any version that allows it to be provided.

To take into account the unavailability of a part of the services associated to a component, the notion of operating modes is introduced. For example, three modes are associated to a valve: *normal*, *blocked_on*, *blocked_off*. In the non-faulty case, the operating mode is *normal*. The *valve_open* and *valve_close* services are available. The operating mode of the valve is *blocked_on* (resp. *blocked_off*) if the valve is blocked in the open (resp. closed) position, and only the service *valve_close* (resp. *valve_open*) is available. Due to the unavailability of a part of the system component services, the achievement of the nominal system objectives may become impossible. The designer may have to implement degraded objectives to increase the system's fault tolerance capability. The system can in this case continue its operations with a reduced but tolerable performance. Nevertheless, the normal system behaviour model is no longer valid. A new model has to be calculated in order to check whether or not the system can continue to meet its objectives. For this reason, in the next section, we propose a self-updating model mechanism that produces the real behavioural model of the system for each observation time.

3. The Self-Updating Modelling

The paper focuses on faults corresponding to the complete loss of a service provided by an actuator. Because a fault of this type totally changes the system's physical structure, the normal system behaviour model is no longer valid. Thus, a flexible system model mechanism is required to automatically update the system model. This mechanism has three parts: a database containing basic knowledge about the system, a procedure for identifying the current system

state of the system, and an algorithm to automatically build the system model using the first and second parts.

3.1. Basic Knowledge Database

Basic knowledge is the knowledge required to infer the current system conditions and predict how the system will evolve. This information is the result of the functional analysis, especially the specification of the control objectives, which requires measurement, calculation and action services to give the process variables their expected values.

More formally, let the basic knowledge be:

- $V = \{v_1, v_2, ..., v_n\}$, the list of the process variables to control,
- $Y_{v_j} = \left\{y_1^{v_j}, y_2^{v_j}, ..., y_x^{v_j}\right\}$, the discrete set of the possible values of the variable $v_j \in V$,
- $C = \{c_1, c_2, ..., c_m\}$, the system components,
- $A_{v_j} = \left\{a_1^{v_j}, a_2^{v_j}, ..., a_q^{v_j}\right\}$, the list of the control actions that can be used by a controller to modify the value of the variable v_j,
- $M_{c_j} = \left\{m_1^{c_j}, m_2^{c_j}, ..., m_r^{c_j}\right\}$, the list of the operating modes for the component c_j,

Let $O = \bigcup_{j=1}^{n} Y_{v_j} + \bigcup_{i=1}^{n} A_{v_i}$ be the set of the observation atoms (i.e: an observation atom is a possible value for a measured output or a control input).

Let $D = \bigcup_{x=1}^{m} M_{c_x}$ be the set of the diagnosis atoms (i.e.: a diagnosis atom is a possible operating mode for a component), where m is the number of components.

A control action a is then defined by two elements:

- $PC(a)$ - a pre-condition that states which observation and diagnosis atoms must hold true before the running of the control action,
- $EF(a)$ - an effect, expressed as a variation direction for the controlled variable, that takes into account the variable's initial value before applying the new control and the values of variables that influence the controlled variable. This effect determines possible values for the controlled variables as the result of the control action.

3.2. Current system state identification

The current system state is given by the subset of the observation and diagnosis atoms that hold true.

Let $O_Y \subseteq \prod_{j=1}^{n} Y_{v_j} = (y_{x_{v_1}}^{v_1}, ..., y_{x_{v_j}}^{v_j}, ..., y_{x_{v_n}}^{v_n})$ - where x_{v_j} denotes an arbitrary number between 1 and the number of elements of the set Y_{v_j}, and $y_{x_{v_j}}^{v_j}$ denotes one of the values of the variable v_j - be the part of the observation vector obtained from the sensor outputs.

Let $O_A \subseteq \prod_{j=1}^{n} A_{v_j} = (a_{x_{v_1}}^{v_1}, ..., a_{x_{v_j}}^{v_j}, ..., a_{x_{v_n}}^{v_n})$ - where x_{v_j} denotes an arbitrary number between 1 and the number of elements of the set A_{v_j}, and $a_{x_{v_j}}^{v_j}$ denotes one of the control action applied on the variable v_j - be the part of the observation vector obtained from the actuator inputs.

Let $D \subseteq \prod_{j=1}^{m} M_{c_j} = (m_{x_{v_1}}^{c_1}, ..., m_{x_{v_j}}^{c_j}, ..., m_{x_{v_n}}^{c_n})$ - where x_{v_j} denotes an arbitrary number between 1 and the number of elements of the set M_{c_j}, and $m_{x_{v_j}}^{c_j}$ denotes one of the modes of the component c_j - be the diagnosis vector. The components of this vector are determined by an appropriated diagnosis module which is not described in this paper but which is presented in [25], [26].

As a result, the current system state is given by the triplet:

$$\begin{aligned} s_0 &= (O_Y(0), O_A(0), D_(0)) \\ &= \left(y_{x_{v_1}}^{v_1}(0), \ldots, y_{x_{v_n}}^{v_n}(0), a_{x_{v_1}}^{v_1}(0), \ldots, a_{x_{v_n}}^{v_n}(0), m_{x_{v_1}}^{c_1}(0), ..., m_{x_{v_n}}^{c_n}(0)\right) \end{aligned}$$

where $y_{x_{v_1}}^{v_1}(0), \ldots, y_{x_{v_n}}^{v_n}(0)$ are the values of the measured outputs, $a_{x_{v_1}}^{v_1}(0), \ldots, a_{x_{v_n}}^{v_n}(0)$ are the values of the control inputs, and $m_{x_{v_1}}^{c_1}(0), ..., m_{x_{v_n}}^{c_n}(0)$ are the operating modes of the system components.

3.3. System Modelling Algorithm

The system modelling algorithm uses the system's basic knowledge to determine the possible successor states from the current system state. It returns a graph $G(S, T, s_0)$ as the updated model, where:

- s_0 is the current system state, given directly from observations and diagnoses,
- $S = \{s_i\}$ is the finite set of states reachable from s_0, with each state s_i being defined as the current state from the list of observation and diagnosis atoms that holds true for this state: $s_i = (O_Y(i), O_A(i), D(i))$,
- $T = \{t_{ij}\}$ is a set of transitions, each of which is defined by $t_{ij} = \{s_i, s_j, a_i\}$ where s_i is the origin state, s_j is the destination state, and $a_i \in \bigcup_{j=1}^{n} A_{v_j}$ is a control action, such as s_i checks $PE(a_i)$ and s_j checks $EF(a_i)$.

This graph can be automatically generated by Algorithm 1. From the current state s_0, the algorithm searches the control actions that can be applied to s_0 among the global control action set $\bigcup_{i=1}^{n} A_i$ (i.e: the control actions in which s_0 verifies the pre-condition PE). If such a control action a_i is found, one or several new states are created (lines 6-7). All these states have:

- an identical set of diagnosis atoms to s_0 (line 8), which is determined by a specific diagnosis module, called the diagnoser, at the time that the current system state is identified,
- an identical set of input observation atoms to s_0, excepting the new control action a_i that replaced the corresponding previous control action (line 9-10),
- a set of output observation atoms, which combines the effects of the actions applied to the variables to be controlled (line 11).

The newly found states, if they have not already be found in a previous step, are added to the graph $G(S, T, s_0)$ (lines 12-14). The search process then is iterated until no new state is found (lines 20-25).

Note that a priori the potential state space of the behaviour model is equal to: $\prod_{j=1}^{n} |Y_{v_j}| \times \prod_{j=1}^{n} |A_{v_j}|$ (the notation $|.|$ is used to express the cardinality of a set). But in reality, among all the possible combinations of variable values and control inputs only a few of them have a physical meaning. For example, a combination *pump=on* and *valve=off* is forbidden if the pump is at the input of a pipe where the valve is at the output. On the same idea a combination with a high liquid temperature and a low liquid level may be forbidden for security reasons. One of the key points of our algorithm is that it does not investigate all the potential states but that it only referees on the available control actions, that extremely limits the complexity. In every cases, the size of the generated behaviour model stays enough small to be easily processed by the classical model checking algorithms.

4. Model Verification

4.1. The model checking requirements

The diagnoser identifies the current system state $s_0 = (O_Y(0), O_A(0), D(0))$. According to Algorithm 1, the successor states of the current system state can be predicted. A fault is detected when a discrepancy exists between the predicted behaviour and the behaviour really observed. The diagnoser's aim is to detect the fault (i.e., a discrepancy exists), to locate the faulty component (e.g., the valve is faulty) and to identify the kind of fault (e.g., the valve is blocked in the open position). This allows the algorithm to update the component's operating modes and the list of the available control actions to act on the process variables.

Algorithm 1 Model Generation

Require: s_0, $\bigcup_{j=1}^{n} A_{v_j}$, $\bigcup_{j=1}^{n} Y_{v_j}$

1: $S \leftarrow \{s_0\}$, $T \leftarrow \oslash$, $test\_states \leftarrow \{s_0\}$, $new\_list \leftarrow \oslash$, $again \leftarrow true$
2: **while** $again$ **do**
3: **for all** $s_{orig} \in test\_states$ **do**
4: **for all** $i = 1$ to n **do**
5: **for all** $a_i \in A_{v_i}$ **do**
6: **if** s_{orig} checks $PC(a_i)$ **then**
7: create a new state s_{dest} such as:
8: $m_{dest_j} = m_{orig_j} \quad \forall j \in [1, m]$
9: $a_{dest_j} = a_{orig_j} \quad \forall j \in [1, n] - \{i\}$
10: $a_{dest_i} = a_i$
11: y_{dest_j} checks $EF(a_j) \quad \forall j \in [1, n]$
12: **if** $t(s_{orig}, s_{dest}, a_i) \notin T$ **then**
13: $new\_list \leftarrow new\_list + \{s\}$
14: $T \leftarrow T + \{t(s_{orig}, s_{dest}, a_i)\}$
15: **end if**
16: **end if**
17: **end for**
18: **end for**
19: **end for**
20: **if** $new\_list \neq \oslash$ **then**
21: $test\_states \leftarrow new\_list$
22: $new\_list \leftarrow \oslash$
23: **else**
24: $again \leftarrow false$
25: **end if**
26: **end while**
27: **return** $G(S, T, s_0)$

The mechanisms on which the diagnoser is based are not described in this paper. Some ideas about building this diagnoser have been given in papers [25], [26].

Once the updated model is obtained (by running Algorithm 1 with the new system conditions as input), the next step is to check whether or not the new model satisfies the system objectives. To this end, the model checking approach translates the updated system model into a labelled transition model, called the execution structure, on which the correctness properties, expressed as temporal logic formulas, are verified by means of specific algorithms [27], [20]. The execution structure is, in our case, the updated model $G(S, T, s_0)$ to which a finite set of logical atoms $L = O_Y \cup O_A \cup D$ is added.

More formally, the execution structure and a path in the execution structure can be described through the following definitions:

Definition 1 (The execution structure). *The execution structure is the state transition graph $M(S, L, T, s_0)$ where:*

- *$L = O_Y \cup O_A \cup D$ is a finite set of logical atoms describing the possible values for the system inputs and outputs, and the possible operating mode for the system components,*
- *S is the state set. Each state $s \in S$ is labelled with a set of atomic propositions $L(s)$ which contains all atoms true in that state, which means that if p is an atomic proposition, then p is true at a state s if and only if p labels s (i.e. p is an element of $L(s)$).*
- *T is the transition set defined as for the system model $G(S, T, s_0)$.*
- *s_0 is the initial state.*

Definition 2 (Path in an execution structure). *A path in an execution structure M in an infinite state sequence $\pi = s_0, s_1, ...$ such that for every $i \geq 0$, $(s_i, s_{i+1}) \in T$. π^i is used to denote the suffix of π starting at s_i.*

This execution structure can then check whether or not a system objective expressed as a sequence of events to observe by the mean of a temporal logic formula, can be achieved. This is the basic idea of the model checking technique.

4.2. The specification of system objectives

Many temporal logics have been defined and studied in the literature. In temporal logic, the truth values of a proposition are not always constant in time. Temporal operators are used to express propositions such as: "The level is *always* equal to 20.", "The level will *eventually* be equal to 20.", "The level will be equal to 20 *until* the valve is opened.". Linear temporal logic (e.g., LTL [28]) reasons about only one possible future for the evolution of a property. Branching time logic (e.g., CTL [29]) assumes that a property might evolve differently in the future if there is an unpredictable environment. For example, such properties as "There is a possibility that the level will stay equal to 20 forever." or "There is a possibility that the level will decrease." can be used to

express different future executions, by taking into account the fact we do not know whether or not the valve will ever open.

The systems we chose to study are partially non-deterministic since the effects of a control action are linked to the current state and system dynamic. Furthermore, these systems are not always designed to achieve the final objectives expressed as final states to be reached. Objectives may be expressed as potentially cyclic execution sequences, such as maintaining a property, achieving an objective periodically or within a number of steps after the request was made, or achieving several objectives in sequence. Branching time logic is quite appropriate for formulating such objectives and integrating the part of non-determinism since it allows the expression of multiple execution possibilities. As the system model is based on a state graph, a temporal logic that refers to "states" rather than "actions" (e.g., TLA [30], ACTL [31]) is preferable. For this reason, we chose the computation tree logic CTL* [32] to formulate the system objectives.

Clarke et al describes CTL*, and its use in model checking, in their texbook [20]. The main definitions and concepts of CTL* and model checking are reviewed below. Readers already familiar with these definitions and concepts can skip directly to section 5.

4.2.1. The CTL semantic*

Let P be the set of the atomic propositions and $p \in P$ an element of P. According to Emerson's work [27], CTL* formulas are built from P using "path quantifiers" and "temporal operators".

- *Path quantifiers* are used in a given state to specify that all or some of the paths starting at that state have certain properties. Two types of path quantifiers are possible:
  - A is a *universal* path quantifier, meaning that certain properties hold true on all paths starting from a given state.
  - E is an *existential* path quantifier, meaning that certain properties hold true on some paths starting from a given state.
- As temporal logic does not explicitly express time as quantity, five *temporal operators* are introduced to describe path properties:
  - X (next time) requires that a property hold true in the path's second state.
  - F (eventually or in the future) is used to affirm that a property will hold true at some states on the path.
  - G (always or globally) specifies that a property holds true at every state on the path.
  - U (until) states that a property holds true if there is a state on the path in which the second property holds true and if the first property holds true at every previous state on the path.

Figure 2: Examples of CTL* formulas [33].

– R (release) requires that the second property holds true along the path, up to and including the first state in which the first property holds, although the first property is not required to hold true in the future.

There are two types of formulas in CTL*: *state formulas*, which are true in a specific state, and *path formulas*, which are true along a specific path. If f is a state formula, the notation $M, s \models f$ means that f holds true at state s in the execution structure M. Similarly, if f is a path formula, the notation $M, \pi \models f$ means that f holds true along path π in M. The relation $\models$ is defined inductively as follow (assuming that f_1, f_2 are state formulas and g_1, g_2 are path formulas):

- $M, s \models p \Leftrightarrow p \in L(s)$ (p is true in the current state)
- $M, s \models \neg f_1 \Leftrightarrow M, s \not\models f_1$
- $M, s \models f_1 \vee f_2 \Leftrightarrow M, s \models f_1$ or $M, s \models f_2$
- $M, s \models f_1 \wedge f_2 \Leftrightarrow M, s \models f_1$ and $M, s \models f_2$
- $M, s \models E(g_1) \Leftrightarrow$ there is a path π from s, such that $M, \pi \models g_1$
- $M, s \models A(g_1) \Leftrightarrow$ for every path π from s, $M, \pi \models g_1$
- $M, \pi \models f_1 \Leftrightarrow s$ is the first state of π and $M, s \models f_1$
- $M, \pi \models X(g_1) \Leftrightarrow M, \pi^1 \models g_1$
- $M, \pi \models F(g_1) \Leftrightarrow$ there is a $k \geq 0$ such that $M, \pi^k \models g_1$
- $M, \pi \models G(g_1) \Leftrightarrow$ for all $i \geq 0$, $M, \pi^i \models g_1$
- $M, \pi \models g_1 U g_2 \Leftrightarrow$ there is a $k \geq 0$ such that $M, \pi^k \models g_2$ and for all $0 \leq j < k$, $M, \pi^j \models g_1$
- $M, \pi \models g_1 R g_2 \Leftrightarrow$ for all $j \geq 0$, if for every $i < j$ $M, \pi^i \not\models g_1$ then $M, \pi^j \models g_2$

The figure 2 gives some examples of CTL* formulas. Each computation tree has the state s_0 as its root [33].

4.2.2. CTL objective formulation*

In order to make easier their translation into CTL* formulas, the system objectives may be split into different properties to checked. According the type of studied systems, a such classification may make appeared:

- reachability or liveness objectives, such as $EF(g)$, which requires that the system may be able to reach desired states where g holds true, and $AF(g)$, which requires that the system will be guaranteed to reach those desired states. The traffic light will turn *green* is an example of liveness objective.

- safety objectives, such as $AG(\neg g)$, which means g must absolutely be avoided, and $EG(\neg g)$, which means that an attempt must be made to avoid g. The lights in cross directions are never *on* at the same time is an example of a safety property for a traffic light controller.

- maintainability objectives, such as $AG(g)$, which means g must be maintained, and $AF(AG(g))$, which means that the system will always reach some future state from which g can be permanently maintained. Any regulation system makes appeared maintainability objective.

. A such decomposition makes easier the written of CTL* formulas for any engineer and brings him an important help to make sure nothing forgotten. He only has to check that all the constraints are correctly extract from the system requirements.

The purpose of our paper is to evaluate the possibilities the system has, or does not have, to carry out its objectives, in faulty situations. The existence of solutions to continue the system operation is based on the existence of multiple versions of the same service, which characterizes the system reconfigurability property. Since control actions are modelled for different outcomes that cannot be predicted at the time of execution (i.e., it is impossible for the system to know a priori which of the different possible outcomes will actually take place), different reconfiguration results may be obtained. For instance, a reconfiguration might guarantee that an objective will be accomplished, or might just provide the possibility of success. CTL* logic allows the differences in these results to be described and defined. For example, the *strong objective* $AF(g)$ means that the reconfiguration guarantees the accomplishment of the desired objectives, while the *weak objective* $EF(g)$ expresses that reconfiguration only has a chance of success.

Automatic reconfiguration is a complex procedure during which the objectives can be changed at the moment that the fault report is received. If nominal objectives cannot continue to be fulfilled, the solution of carrying out the degraded objectives has to be evaluated. For a desired property g, the temporal objective would be $AG(g)$ (i.e., g always holds true) in the normal operating mode. If there is a fault that causes g to deviate from its desired value, then reconfiguration will correct this deviation and will keep g within its desired value range. But in a non-deterministic system, a control action sent by the reconfiguration procedure cannot be guaranteed to produce the desired effects (i.e.,

the original objective cannot be guaranteed). This situation can be described as $EF(AG(g))$, which means that g will eventually be accomplished in some future state from which g will be permanently maintained. However, this does not satisfy the requirements of some high-security systems. Thus, $EF(AG(g))$ must be changed into a strong solution, such as $AF(AG(g))$. If the execution structure satisfies the objective $AF(AG(g))$, then reconfiguration will be successful in spite of non-determinism.

4.3. The model checking

Let Ψ be a objective expressed in CTL* logic. Let $M(S, L, T, s_0)$ be the execution structure. The model checking task aims to determine which states in S satisfy Ψ. To accomplish this, the formula Ψ is first decomposed into sub-formulas that make only the connectives $\neg, \wedge, \bot, EX, EG$ and EU appear, using the following equivalences [20]:

- $AX(f) = \neg EX(\neg f)$
- $EF(f) = E(trueUf)$
- $AG(f) = \neg EF(\neg f)$
- $AF(f) = \neg EG(\neg f)$
- $A[fUg] \equiv \neg E[\neg gU(\neg f \wedge \neg g)] \wedge \neg EG(\neg g)$
- $A[fRg] \equiv \neg E[\neg fU\neg g]$
- $E[fRg] \equiv \neg A[\neg fU\neg g]$

The labelling operation is then processed. This operation determines the set $label(s)$ of the sub-formulas of Ψ that are all true in s. This operation is repeated for each state of M, starting with the smallest sub-formulas and working recursively towards Ψ. Initially, $label(s)$ is just $L(s)$. The algorithm then processes the sub-formulas with nested CTL* operators. When the algorithm terminates, the result is the set of states of the model that satisfy the formula. If this set includes the initial state s_0, then M satisfies Ψ.

Some examples are given below to describe how states are labelled.

If Ψ is:

1. $\bot$: then no states are labelled with $\bot$.
2. p: then label s with p if $p \in L(s)$.
3. $\Psi_1 \wedge \Psi_2$: label s with $\Psi_1 \wedge \Psi_2$ if s is already labelled both with Ψ_1 and Ψ_2.
4. $\neg\Psi_1$: label s with $\neg\Psi_1$ if s is not already labelled with Ψ_1.
5. $EG(\Psi_1)$:
   - Label all the states with $EG(\Psi_1)$,
   - If any state s is not labelled with Ψ_1, delete the label $EG(\Psi_1)$,

- **repeat**
  delete the label $EG(\Psi_1)$ from any state if none of its successors is labelled with $EG(\Psi_1)$
  until there is no change.

6. $E(\Psi_1 U \Psi_2)$
   - If any state s is labelled with Ψ_2, label it with $E(\Psi_1 U \Psi_2)$
   - **repeat**
     label any state with $E(\Psi_1 U \Psi_2)$ if it is labelled with Ψ_1 and at least one of its successors is labelled with $E(\Psi_1 U \Psi_2)$
     until there is no change.
7. $EX(\Psi_1)$: label any state with $EX(\Psi_1)$ if one of its successors is already labelled with Ψ_1.
8. $AF(\Psi_1)$
   - If any state s is labelled with $AF(\Psi_1)$, label it with $AF(\Psi_1)$
   - **repeat**
     label any state with $AF(\Psi_1)$ if all successor states are labelled with $AF(\Psi_1)$
     until there is no change.

These eight sub-algorithms can be combined to deal recursively with the different formulas. For example, the verification of the formula $S(AF(AG(x)))$ will be processed in three steps: 1- $S(x)$, 2- $S(AG(x))$, and 3- $S(AF(AG(x)))$. Applied to the reconfigurability analysis of a system, this method make it possible to determine whether or not a faulty system has the potential to continue its operation without intolerable performance losses or with reduced specifications. Let now apply the proposed methodology on a didactic example.

5. Example: The Two Tank System (TTS)

5.1. A description of our TTS

We chose the TTS to illustrate our method because this example (or similar ones) has been used to illustrate different approaches in fault diagnosis and fault tolerant control [2], [34], [35], [36]. This example is related to a level-regulation process involving two identical connected tanks (see Fig. 3). The main objective of the TTS is to provide a continuous water flow Q_O to a consumer via an outlet valve V_O, located at the bottom of tank T_2. To accomplish this objective, pump P_1 fills tank T_1 up to a nominal water level of 50 cm. The flow in tank T_2 is kept at a level of 10 cm via valve V_1 placed in tank T_1 on a connecting pipe at level 30 cm. Valve V_2, placed in tank T_1 on a connecting pipe at level 0 cm, is always closed in the nominal case and is used as a backup valve when V_1 is faulty.

The nominal objective is to regulate the level in tank T_1 up to a water level of 50 cm. In the case in which the nominal objective cannot be fulfilled, users accept a regulation to a water level just under the valve V_1 location (i.e., 30 cm).

Figure 3: The two tank process.

The available process measurements are the water levels l_1 for tanks T_1, and l_2 for tank T_2, given respectively by the sensors L_1 and L_2. For the nominal case, a controller C_1 turns off the pump P_1 when l_1 is 50 cm (degraded case: 30 cm) and turns it *on* when l_1 reaches 45 cm (degraded case: 25 cm). Another controller C_2 turns *on* valve V_1 (or V_2) if l_2 falls to 9 cm and turns it off when level rises to 11 cm.

There were many reasons for choosing the TTS example for our paper, other than the one stated above. First, multi-tank systems are widely used in industrial domains as diverse as chemistry, production and power plants. Second, many physical processes include the inflow and outflow of materials and thus can be modelled by multi-tank systems. A complex system can be decomposed into many of these small basic systems for analysis. Third, this example allows us to specify nominal and degraded objectives and illustrates some interesting proprieties, such as the cycle execution and the non-determinism. In fact, the TTS regulation process requires a cyclic execution of services that makes it possible to increase the levels and then to decrease them. The flow between the two tanks varies according to the different water levels in each tank. Thus, during the regulation process, a single valve action results in different water level changes in each tank. This is a non-deterministic property that has to be formulated correctly.

5.2. The functional viewpoint

From the TTS description, the nominal control objective is "to regulate the level in tank T_1 between 45 cm and 50 cm and to regulate the level in tank T_2 between 9 cm and 11 cm". The degraded version of this objective is "to regulate the level in tank T_1 between 25 cm and 30 cm and to regulate the level in tank T_2 between 9 cm and 11 cm". Accomplishing this control objective relies on the services provided by the TTS components: the two valves V_1 and V_2, pump P_1, tanks T_1 and T_2, sensors L_1 and L_2 and controllers C_1 and C_2. The sensor L_i provides a measurement service: L_i_level. The valve V_i provides two actuation services: V_i_open and V_i_close. The pump P_1 provides two services: P_1_open and P_1_close. The tank T_i provides a storage service T_i_store. The controller C_i provides a calculation service according a regulation algorithm. Several operating modes can be associated to the TTS components: {*normal, non normal*} for the tanks, the controllers and the sensors and {*normal, blocked_on, blocked_off*} for the valves and the pump. This functional analysis supports the definition of the system's behaviour model.

5.3. The system model

The basic knowledge required by the TTS to infer the current system conditions and predict the system's evolution is as follow:

- The list of the process variables to control is $V = \{l_1, l_2\}$
- The possible values for the variable l_1 are :
  $Y_{l_1} = \{[0, 25[, 25,]25, 30[, 30,]30, 45[, 45,]45, 50[, 50,]50, 60]\}$
- The possible values for the variable l_2 are: $Y_{l_2} = \{[0, 9[, 9,]9, 11[, 11,]11, 60]\}$
- The system components are: $C = \{P_1, V_1, V_2\}$
- The component that can be controlled to act on l_1 is: $C_{l_1} = \{P_1\}$
- The components that can be controlled to act on l_2 are: $C_{l_2} = \{V_1, V_2\}$
- The possible actions to modify l_1 are : $A_{l_1} = \{P_1\_on, P_1\_off\}$
- The possible actions to modify l_2 are: $A_{l_2} = \{V_1\_on, V_1\_off, V_2\_on, V_2\_off\}$
- The operation modes for the component P_1 are:
  $M_{P_1} = \{ok(P_1), stuck\_on(P_1), stuck\_off(P_1)\}$
- The operation modes for the component V_i, where $i = 1; 2$ are:
  $M_{V_i} = \{ok(V_i), stuck\_on(V_i), stuck\_off(V_i)\}$

We assumed that the TTS sensors, controllers and tanks cannot fail. For this reason, they do not appear in the basic knowledge.

Based on the basic knowledge description,

- the set of the observation atoms is:
  $O = \{l_1 \in [0, 25[, l_1 = 25, l_1 \in]25, 30[, l_1 = 30, l_1 \in]30, 45[, l_1 = 45, l_1 \in]45, 50[, l_1 = 50,$
  $l_2 \in [0, 9[, l_2 = 9, l_2 \in]9, 11[, l_2 = 11, l_2 \in]11, 60], P_1\_on, P_1\_off, V_1\_on, V_1\_off,$
  $V_2\_on, V_2\_off\}$
- the set of the diagnosis atoms is:
  $D = \{ok(P_1), stuck\_on(P_1), stuck\_off(P_1), ok(V_1), stuck\_on(V_1), stuck\_off(V_1),$
  $ok(V_2), stuck\_on(V_2), stuck\_off(V_2)\}$

Each control action is specified in term of pre-conditions and effects as follow where the sign '↑' (resp. '↓', '→') means that the level rises (resp. falls, keeps the same value).

- $P_1\_off :=$
  - case $\neg stuck\_on(V_1)$
    $PC(P_1\_off) = P_1\_on \wedge (l_1 = 50)$
    $EF(P_1\_off) = l_1 \rightarrow$ if V_1_close
    $EF(P_1\_off) = l_1 \downarrow$ if V_1_open
  - case $stuck\_on(V_1)$
    $PC(P_1\_off) = P_1\_on \wedge (l_1 = 30))$
    $EF(P_1\_off) = l_1 \rightarrow$ if V_2_close
    $EF(P_1\_off) = l_1 \downarrow$ if V_2_open

- $P_1\_on :=$
  - case $\neg stuck\_on(V_1)$
    $PC(P_1\_on) = P_1\_off \wedge (l_1 = 45)$
    $EF(P_1\_on) = l_1 \downarrow$ if V_1_open
    $EF(P_1\_on) = l_1 \uparrow$ if V_1_close
  - case $stuck\_on(V_1)$
    $PC(P_1\_on) = P_1\_off \wedge (l_1 = 25))$
    $EF(P_1\_on) = l_1 \downarrow$ if V_2_open
    $EF(P_1\_on) = l_1 \uparrow$ if V_2_close
- $V_1\_close :=$

  $PC(V_1\_close) = V_1\_open \wedge (l_2 = 11)$
  $EF(V_1\_close) = l_2 \downarrow$
- $V_1\_open :=$
  - case $\neg stuckC(P_1)$
    $PC(V_1\_open) = V_1\_close \wedge (l_2 = 9)$
    $EF(V_1\_open) = l_2 \uparrow$
  - case $stuckC(P_1)$
    $PC(V_1\_open) = V_1\_close \wedge (l_2 = 9))$
    $EF(V_1\_open) = l_2 \uparrow$ if $(l_1 > 30)$
    $EF(V_1\_open) = l_2 \downarrow$ if $(l_1 < 30)$
- $V_2\_open :=$
  - case $\neg stuckC(P_1)$
    $PC(V_2\_open) = V_2\_close \wedge stuck\_O(V_1) \wedge (l_2 = 9)$
    $EF(V_2\_open) = l_2 \uparrow$
  - case $stuckC(P_1)$
    $PC(V_2\_open) = V_2\_close \wedge stuck\_O(V_1) \wedge (l_2 = 9))$
    $EF(V_2\_open) = l_2 \uparrow$ if $(l_1 > l_2 > 0)$
- $V_2\_close :=$

  $PC(V_2\_close) = V_2\_open \wedge (l_2 = 11)$
  $EF(V_2\_close) = l_2 \downarrow$

5.4. Generating the model

The system's behaviour model can be automatically generated from the system's basic knowledge and from the identification of the current system state. Suppose that the diagnoser identifies the system's current state as: $s_0 = \{l_1 \in]45, 50[, l_2 = 11, P_1\_close, V_1\_open, ok(P_1), ok(V_1)\}$.
In this case, the system is not faulty since the diagnosis vector is $D_N = \{ok(c_j)\}\, \forall c_j \in C_m$. The corresponding system behaviour model is the

Figure 4: The normal behaviour model.

normal behaviour model (Figure 4). This model can be automatically generated from s_0 by running Algorithm 1. Algorithm 1 receives as input the state s_0, denoted state 1 on the graph presented in Figure 4. Among the control actions in $A_{l_1} \cup A_{l_2}$, only V_1_close has its pre-condition verified by s_0. This control action takes the place of V_1_open. The list of the input observation atoms of the successor states of s_0 is P_1_close, V_1_open. The list of the output observation atoms of the successor states of s_0 is obtained from the possible combinations of the effects of the control actions P_1_close, V_1_open. These effects are "to maintain l_1" for P_1_close and "to decrease l_2" for V_1_open. State 2 on the graph given in Figure 4 was found in this way. A state with $l_1 \in]45, 50[, l_2 \in]9, 11[$ is not taken into account as such a state leads to no new control actions. Successors states from s_2 are determined in the same way. The system model does not need to be recalculated when there is no change in the diagnosis vector D.

5.5. Formulating the TTS objectives

In the normal operating mode, the objective of the TTS is to maintain the tank T_2 level between 9 cm and 11 cm and to try to maintain the tank T_1 level between 45 cm and 50 cm. This temporal objective can be expressed as $AG(9 \leq l_2 \leq 11) \wedge EG(45 \leq l_1 \leq 50)$. The nominal objective is to regulate the level in tank T_1 up to a water level of 50 cm. In the case in which the nominal objective cannot be fulfilled, users accept a revised objective that regulates the water level in tank T_1 just under the valve V_1 location (i.e., 30 cm). This degraded objective is acceptable only in faulty situations and can be expressed by the CTL* formula: $AF(AG(9 \leq l2 \leq 11) \wedge EG(25 \leq l_1 \leq 30))$.

The system is assumed to be well designed and thus able to fulfil the nominal objective in non-faulty situations. This hypothesis can nevertheless be checked by testing the formula $AG(9 \leq l_2 \leq 11) \wedge EG(45 \leq l_1 \leq 50)$ on the execution structure corresponding to the system's nominal model (see Figure 4). In faulty situation, the reconfigurability analysis procedure first tests the nominal objective, expressed as $AF(AG(9 \leq l2 \leq 11) \wedge EG(45 \leq l_1 \leq 50))$. If the evaluation result shows that the nominal objective cannot be fulfilled, the degraded objective, expressed as $AF(AG(9 \leq l2 \leq 11) \wedge EG(25 \leq l_1 \leq 30))$ is tested.

5.6. The model checking procedure

Suppose the current state of the system is the state noted 1 on the figure 4. From this state, the model generation algorithm predicts that the control

Figure 5: Model obtained when valve V_1 is stucked in the opened position.

action $V1\_close$ will be applied and that the system will evolve to the state 2. Consequently to the running of this new control action, the level l_2 must decrease. If this level is increasing instead of decreasing a fault is detected. The fault detection time is linked to the system dynamic. If a priori any component may become faulty at any time, a fault sign usually manifests after attempting to apply an action on the system. To continue with our example, if the predicted state noted 2 in the figure 4 is not observed when the control action $V1\_close$ is run, it is probably because this action has not been correctly realised. Consequently, the fault localisation time can be reduced by testing as first fault mode, a fault mode corresponding to the last action applies on the system (i.e. $V1\_stuck\_on$ in our example).

The corresponding revised system model obtained when the diagnoser detects that valve V_1 is blocked in the opened position is given in Figure 5. State 1 is the current system state at the time the diagnoser detects the fault. This model is the execution structure for the model checking procedure. The nominal objective to be checked is expressed by the CTL* formula $AF(AG(9 \leq l2 \leq 11) \wedge EG(45 \leq l1 \leq 50))$. The current state (state 1) obviously does not check this CTL* formula, but the question remains "Will the nominal objective be fulfilled in the future ?". To answer to this question, the CTL* formula is tested as proposed in section 4.3.

Let be f the abbreviation for $9 \leq l2 \leq 11$ and g be the abbreviation for $45 \leq l1 \leq 50$. The CTL* formula $AF(AG(9 \leq l2 \leq 11) \wedge EG(45 \leq l1 \leq 50))$ is first re-written in terms of the basic connectives $\neg, \wedge, \bot, EX, EG$ and EU, using the equivalences given in section 4.3. The CTL* formula becomes $AF(\neg EF(\neg f) \wedge EG(g))$. The labelling operation is then processed, dealing recursively with the different sub-formulas. Let $S(\psi)$ denote the set of all states labelled with the sub-formula ψ. The processing steps are:

1. $S(g) = \{1\}$. Only state 1 shown on the graph given in Figure 5 verifies $45 \leq l1 \leq 50$.
2. $S(EG(g)) = \{\}$. In order to calculate $S(EG(g))$, the states of $S(g)$ are first labelled with $EG(g)$, and then the label $EG(g)$ is deleted from any state if none of it successors is labelled with $EG(g)$. This deletion procedure is repeated until there is no change.
3. $S(f) = \{3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$. There are the states that satisfy $9 \leq l2 \leq 11$.
4. $S(\neg f) = \{1, 2\}$
5. $S(EF(\neg f)) = S(E(trueUf) = \{1, 2\}$
6. $S(\neg EF(\neg f)) = \{3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$
7. $S(\neg EF(\neg f) \wedge EG(g)) = \{\}$

8. $S(AF(\neg EF(\neg f) \wedge EG(g))) = \{\}$

This result shows that the nominal objective cannot be met when the valve V_1 is blocked in the open position. The second step of the reconfiguration procedure tests the degraded objective, expressed as $(AG(9 \leq l2 \leq 11) \wedge EG(25 \leq l1 \leq 30))$. Re-written in terms of basic connectives, the revised objective is $AF(\neg EF(\neg f) \wedge EG(g))$, where f stands for $9 \leq l2 \leq 11$ and g stands for $25 \leq l1 \leq 30$. The different steps of the labelling procedure are:

1. $S(g) = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$.
2. $S(EG(g)) = \{\}$.
3. $S(f) = \{3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$.
4. $S(\neg f) = \{1, 2\}$
5. $S(EF(\neg f)) = S(E(trueUf) = \{1, 2\}$
6. $S(\neg EF(\neg f)) = \{3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$
7. $S(\neg EF(\neg f) \wedge EG(g)) = \{3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$
8. $S(AF(\neg EF(\neg f) \wedge EG(g))) = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$. In order to calculate $S(AF(\psi))$, the states of $S(\psi)$ are, first, labelled with $AF(\psi)$, and then the label $AF(\psi)$ is applied to any state that has all its successors are labelled with $AF(\psi)$ until there is no change. State 2 is first found by this way and then state 1.

This result is very strong because it means that the verified objective holds true along every path from any state in the execution structure. Specifically, since this set contains the initial state 1, it can be concluded that, with this execution structure, if the reconfigurable control starts at the moment when the fault is identified and the model is recalculated, the degraded objective $(AG(9 \leq l2 \leq 11) \wedge EG(25 \leq l1 \leq 30))$ is guaranteed to be met at some later time.

6. Conclusion

By proposing a flexible model for expressing the real possibilities of controlling a system when a fault occurs, we bring an important help to operators who have to exactly know, at each time, what they can obtain from the system they supervise. Moreover, these possibilities can be expressed in term of available and unavailable functions, and this kind of information is well adapted to the human reasoning. The flexible model we propose, is built on this functional analysis. It allows to take into account the dynamic aspect required to oversee the system on line, and to evaluate the availability of the services provided by the system components. It takes the form of a state transition graph. Each state groups a set of atoms whose values are true for this state. The atoms are observation atoms (i.e., measured values, control action values) and diagnosis

atoms (i.e., the normal and faulty operation modes of the components). The transitions between the states correspond to control action that can be applied to the system when it is in the origin state to drive it to a destination state. The transition graph is automatically built by an appropriated algorithm each time the values of the diagnosis atoms change. The algorithm requires as input the values of the observation and diagnosis atoms and a description of the control actions that can be provided by the system's components, expressed in the form of a pre-condition and one or more effects. This description is obtained from the functional analysis of the system. The values of the diagnosis atoms are updated by the diagnoser which is not described in this paper. The basic ideas for building the diagnoser are the following. Comparing the states predicted by the flexible model and the states actually observed makes it possible detect faults. Using fault models, expressed as control actions, in terms of preconditions and effects, make it possible to identify the faulty component and the fault mode [25] and [26].

By associating diagnosis and reconfiguration, all built on the same model, we hope to contribute to the development of a global supervisory control system. Indeed, the interest of using the same model is to improve the delay to reconfigure the system when a fault appears. Possible future research is first to implement the global solution for supervising a system in real time and second test our solution on a large system. One of the future applications is the supervision of intelligent autonomous vehicles in the InTraDe project (Intelligent Transportation for Dynamic Environments). This project is financed by European regional development funding through InterregIVB and is supposed to lead to the development of an automatic navigation system for port terminals. In this context, the ultimate objective of our work is to design a supervision system representing the management of the operating modes of a vehicle and giving the conditions for its reconfiguration when a part of its inner components is faulty or when another vehicle with it has to cooperate is not in a nominal operating mode. One of the challenge is to see how the proposed representation can be distributed on a set of subsystems (the vehicles or some parts of a same of vehicle). Global objective has for this to be decomposed into local sub-objectives and local diagnosis and reconfiguration possibilities have to be merged to reconstruct global diagnosis and reconfiguration solutions.

References

[1] M. Staroswiecki, A.-L. Gehin, From control to supervision, Annual Reviews in Control 25 (2001) 1–11.

[2] M. Blanke, M. Kinnaert, J. Lunze, M. Staroswiecki, J. Schröder, Diagnosis and Fault-Tolerant Control, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[3] R. Isermann, Fault-Diagnosis Systems: An Introduction from Fault Detection to Fault Tolerance, Springer, 2005.

[4] V. Venkatasubramanian, R. Rengaswamy, K. Yin, S. N. Kavuri, A review of process fault detection and diagnosis: Part i: Quantitative model-based methods, Computers & Chemical Engineering 27 (3) (2003) 293–311.

[5] V. Venkatasubramanian, R. Rengaswamy, S. N. Kavuri, A review of process fault detection and diagnosis: Part ii:qualitative models and search strategies, Computers & Chemical Engineering 27 (3) (2003) 313–326.

[6] V. Venkatasubramanian, R. Rengaswamy, S. N. Kavuri, K. Yin, A review of process fault detection and diagnosis: Part iii: Process history based methods, Computers & Chemical Engineering 27 (3) (2003) 327–346.

[7] A.-L. Gehin, M. Staroswiecki, Reconfiguration analysis using generic component models, IEEE Transactions on Systems Man and Cybernetic-Part A 38 (3) (2008) 575–583.

[8] M. Blanke, M. Staroswiecki, E. Wu, Concepts and methods in fault-tolerant control, in: Proceedings of American Control Conference, Washington DC, USA, 2001, pp. 2606–2620.

[9] Y. Zhang, J. Jiang, Bibliographical review on reconfigurable fault-tolerant control systems, Annual Reviews in Control 32 (2008) 229–252.

[10] P. A. Ioannou, J. Sun, Robust Adaptive Control, Prentice Hall, 1995.

[11] K. Zhou, J. C. Doyle, Essentials of robust control, Prentice Hall, 1998.

[12] Y. W. Liang, D. C. Liaw, T. C. Lee, Reliable control of nonlinear systems, IEEE Transactions on Automatic Control 45 (4) (2000) 706–710.

[13] S. Ye, Y. Zhang, C.-A. Rabbath, X. Wang, Y. Li, An lmi approach to mixed $h2h\infty$ robust fault-tolerant control design with uncertainties, in: ACC'09: Proceedings of the 2009 conference on American Control Conference, IEEE Press, Piscataway, NJ, USA, 2009, pp. 5540–5545.

[14] J. S. H. Niemann, An architecture for fault tolerant controllers, International Journal of Control 78 (14) (2005) 1091–1110.

[15] D. Theilliol, C. Join, Y. Zhang, Actuator fault tolerant control design based on a reconfigurable reference input, Int. J. Appl. Math. Comput. Sci. 18 (4) (2008) 553–560.

[16] J. Lunze, J. H. Richter, J. Maciejowki, Reconfigurable fault-tolerant control : A tutorial introduction. discussion. commentary, European Journal of Control 14 (5) (2008) 359–390.

[17] J. Lunze, Control reconfiguration after actuator failures: the generalised virtual actuator, in: Proceedings of the 6th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes (SAFEPROCESS), IFAC, Beijing, China, 2006, pp. 1309–1314.

[18] G. Tao, S. H. Chen, S. M. Joshi, An adaptive actuator failure compensation controller using output feedback, IEEE Transactions on Automatic Control 47 (3) (2002) 506–511.

[19] S. Chen, G. Tao, S. M. Joshi, On matching conditions for adaptive state tracking control of systems with actuator failures, Transactions on Automatic Control 47 (3) (2002) 473–478.

[20] E. M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 1999.

[21] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, Systems and Software Verification. Model-Checking Techniques and Tools, Springer, 2001.

[22] C. Baier, J. P. Katoen, K. Larsen, Principles of Model Checking, MIT Press, 2008.

[23] K. Havelund, M. Lowry, J. Penix, Formal analysis of a space-craft controller using spin, IEEE Transactions on Software Engineering 27 (8) (2001) 749–765.

[24] L. Molnar, S. Veres, System verification of autonomous underwater vehicles by model checking, in: IEEE OCEANS conference, Bremen, Germany, 2009.

[25] H. Hu, A.-L. Gehin, M. Bayart, An extended qualitative multi-faults diagnosis from first principles i: Theory and modelling, in: Proceeding of the 48th IEEE Conference on Decision and Control, Shangai, 2009.

[26] H. Hu, A.-L. Gehin, M. Bayart, An extended qualitative multi-faults diagnosis from first principles ii: Algorithm and case study, in: Proceeding of the 48th IEEE Conference on Decision and Control, Shangai, 2009.

[27] E. A. Emerson, Temporal and modal logic, in: Handbook of Theoretical Computer Science (Vol. B): formal models and semantics, MIT Press, 1991, pp. 995–1072.

[28] Z. Manna, A. Pnueli, The temporal logic of reactive and concurrent systems, Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[29] E. Clarke, E. A. Emerson, A. P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Trans. Program. Lang. Syst. 8 (2) (1986) 244–263.

[30] L. Lamport, The temporal logic of actions, ACM Transactions on Programming Languages and Systems 16 (1994) 872–923.

[31] R. De Nicola, F. Vaandrager, Action versus state based logics for transition systems, in: Proceedings of the LITP spring school on theoretical computer science on Semantics of systems of concurrent processes, Springer-Verlag New York, Inc., New York, NY, USA, 1990, pp. 407–419.

[32] E. A. Emerson, J. Y. Halpern, "sometimes" and "not never" revisited: on branching versus linear time temporal logic, J. ACM 33 (1) (1986) 151–178.

[33] M. Huth, M. Ryan, Logic in Computer Science: Modeling and Reasoning about Systems, Cambridge University Press, 2004.

[34] K. J. Åström, P. Albertos, M. Blanke, A. Isidori, W. Schaufelberger, Control of Complex Systems, Springer-Verlag, 2001.

[35] D. Theilliol, H. Nouraa, J. C. Ponsarta, Fault diagnosis and accommodation of a three-tank system based on analytical redundancy, ISA Transactions 41 (3) (2002) 365–382.

[36] C. Join, H. Sira-Ramirez, M. Fliess, Control of an uncertain three-tank-system via on-line parameter identification and fault detection, in: Proceeding of the 6th IFAC World Congress on Automatic Control, Prague, 2005.