Designing Systems with Detection and Reconfiguration Capabilities: A Formal Approach

Iulia Dragomir, Simon Iosti, Marius Bozga, Saddek Bensalem

Designing Systems with Detection and Reconfiguration Capabilities: A Formal Approach*

Iulia Dragomir, Simon Iosti, Marius Bozga, and Saddek Bensalem

Univ. Grenoble Alpes, CNRS, Grenoble INP**,VERIMAG, 38000 Grenoble, France

Abstract. The design of functionally correct autonomous systems which operate in an unknown environment and that satisfy reliability, availability, maintainability, and safety (RAMS) requirements is a challenge. In this paper we focus on the detection and reconfiguration features these systems must provide. Indeed, evolving in an unknown environment can invalidate the assumptions made during the design phase. In particular, different hardware components might fail and provide erroneous inputs to the system, which will pass in a degraded mode where the expected RAMS do not hold anymore. Such faults need to be detected as early as possible and reconfiguration strategies must be applied to bring the system back into a nominal mode where the RAMS are satisfied. We propose an automated design process based on formal methods to develop Fault Detection, Isolation and Recovery (FDIR) components targeting partially observable timed systems.We describe how to automatically synthesize runtime monitors, design reconfiguration strategies, and obtain full-fledged FDIR components. We illustrate the approach on a case study inspired from autonomous robotics applications.

1 Introduction

Mission- and safety-critical systems must satisfy a plethora of important Reliability, Availability, Maintainability and Safety (RAMS) properties, which is a hard problem to establish at design time. The reason is two-fold: (i) the built systems are very complex and verification techniques do not always scale on real-life applications, and (ii) such systems often work in unknown environments that may not satisfy at execution time the assumptions made at design time. This is the case of autonomous systems that execute in an environment subject to faults and failures. For instance, a hardware component might overheat, which leads the entire system in a degraded mode and where the above mentioned requirements do not hold anymore.

A desired functionality of safety-critical systems in general, and autonomous systems in particular, is to detect and handle systematically and dynamically

* This work has been supported by the HORIZON 2020 PROGRAMME Strategic Research Cluster (SRC) (awards #730080 and #730086).

** Institute of Engineering Univ. Grenoble Alpes

the faults that have occurred. The handling of such faults could be either simple by giving control of the system to a human user or complex by applying predefined strategies for recovery. Autonomous systems usually fall in the latter case, where they should implement without any external intervention complex recovery strategies that aim to bring the system back into a safe state where RAMS hold again. This involved functionality is implemented by *Fault Detection, Isolation and Recovery* (FDIR) *components*, which extend such systems with adaptive and collaborative features. On one hand, the adaptive aspect is inherent to the definition of FDIR components that steer the system operation depending on environment and specific operating conditions. On the other hand, the collaborative aspect arises as various subsystems and FDIR are generally interacting together for achieving a common goal. At system level, such subsystems are components while the goals are maintaining individual RAMS properties. At mission level, goals can be more involved and concern high-level objectives.

An FDIR component runs in parallel with the system, and (i) detects faults as early as possible with respect to their occurrence and (ii) executes a predefined recovery strategy with respect to the detected fault. The extended system can contain one or multiple FDIR components, which can have a monolithic or hierarchical architecture, can be centralized or distributed, or any combination. The process of designing/implementing FDIR components is *ad-hoc*, based on one's full understanding of the system under design, the component to be produced and the system's (possibly textual) specification. This implies considering a large number of faults and failures, their interactions and effects on the system, which raises correctness and completeness questions.

We answer such problems by proposing a methodology based on formal methods to build FDIR components. The aim is to automatically derive correct FDIR components from the design of the system under study with faults, the RAMS requirements it must satisfy and the recovery strategies to be applied in case of faults. We describe how to automatically synthesize runtime monitors for fault detection, design the recovery strategies for controller synthesis and obtain full-fledged FDIR components. Moreover, we tackle the above problems in the context of timed systems and partial observability, where faults cannot be directly detected by the system and not all of the system's actions can be observed. We illustrate the approach and its feasibility on an excerpt of an industrial autonomous robotics application.

Paper structure. We formalize timed systems with partial observability (for the FDIR context) in Section 2. In Section 3 we describe the methodology for designing FDIR components as a subprocess of the general system design. The algorithms for diagnoser synthesis and controller implementation for our definition of timed systems are given and illustrated in Section 4 and Section 5, respectively. We discuss the work related to the methodology as well as the diagnoser and controller synthesis problems in Section 6 before concluding.

2 Timed systems

We consider a system modeled as a network of timed automata (TA) [2]. Before formally defining timed automata, we introduce some notations.

Let X be a finite set of variables called *clocks*. A *clock valuation* is a mapping $\mathrm{v} : X \to \mathbb{R}_+$. We write $\mathbf{0}$ for the valuation of all clocks of X to 0. Given $\delta \in \mathbb{R}_+$, $(\mathrm{v}+\delta)(x) = \mathrm{v}(x)+\delta$. For $r \subseteq X$, $\mathrm{v}[r]$ is the reset of clocks in r, i.e., the valuation defined by $\mathrm{v}[r](x) = \mathrm{v}(x)$ if $x \notin r$ and $\mathrm{v}[r](x) = 0$ otherwise. Let $\Phi(X)$ be the set of convex constraints on X given by the grammar $\varphi ::= \mathit{true} | x < c | x \leq c | x = c | x > c | x \geq c | \varphi \wedge \varphi$, with $c \in \mathbb{Q}_+$. Given a constraint $g \in \Phi(X)$ and a valuation v, we write $\mathrm{v} \models g$ if g is satisfied by the valuation v.

Definition 1 (Timed automaton). *A* timed automaton *(TA) A is a tuple $(L, l_0, X, Inv, \Sigma, E)$ where L is a finite set of locations, $l_0 \in L$ is the initial location, X is a finite set of clocks, $Inv : L \to \Phi(X)$ is a function associating to each location some clock constraint, $\Sigma = \Sigma_o^c \oplus \Sigma_o^u \oplus \Sigma_u^s \oplus \Sigma_u^f$ is a finite set of actions separated into observable/unobservable (denoted with subscript) and controllable/uncontrollable (denoted with superscript) as explained below, and $E \subseteq L \times \Phi(X) \times \Sigma \times 2^X \times L$ is the transition relation.*

A timed automaton is a finite automaton enriched with a set of real-valued clocks that allow to measure time delays. In this computational model, time passes at the same rate for all clocks, i.e., $\dot{x} = 1$. Time elapse is restricted in each location with a clock constraint. A transition, usually denoted by $l \xrightarrow[r]{[g]\ a} l'$, moves from a location l to a location l' by executing an action $a \in \Sigma$. The transition is enabled and can be fired only when the current valuation of clocks satisfies the guard $g \in \Phi(X)$. Besides the executed action, a transition can also perform resets on the specified set $r \subseteq X$ of clocks.

With respect to the definition given in [2], the main difference is the *partial observability condition*. It is modeled by the two types of actions a TA can define: observable actions Σ_o and unobservable actions Σ_u. The observable actions are further refined into controllable ones Σ_o^c and uncontrollable ones Σ_o^u. The controllable observable actions act as "actuators" for the FDIR component, while uncontrollable observable actions act as "sensors" for fault detection. The unobservable actions are also refined into regular ones (also called silent) Σ_u^s and faulty ones Σ_u^f. Silent actions correspond to internal computations often denoted by τ. Fault actions are those that model the different types of faults of a component. Please note that all the above sets are disjoint. By taking $\Sigma_o^c = \emptyset$, $\Sigma_u^s = \{\tau\}$ and $\Sigma_u^f = \emptyset$, we obtain the usual definition of TA with silent actions. The definition from [29,18] is obtained for $\Sigma_o^c = \emptyset$, $\Sigma_u^s = \{\tau\}$ and $\Sigma_u^f = \{f\}$.

The semantics of a timed automaton is a Timed Transition System (TTS). A state of the TA is a pair $(l, \mathrm{v}) \in L \times \mathbb{R}_+^X$ that consists of a discrete location $l \in L$ and the current valuation of all clocks v. The initial state is the pair $(l_0, \mathbf{0})$. From a state (l, v) such that $\mathrm{v} \models Inv(l)$, the TA can progress either by a discrete transition (i.e., an action) or by letting time elapse. The transition relation $\to$ of the corresponding TTS is generated by the following rules:

1. For $a \in \Sigma$, $(l, \mathrm{v}) \xrightarrow{a} (l', \mathrm{v}')$ if $l \xrightarrow[r]{[g]\ a} l'$ such that $\mathrm{v} \models g$, $\mathrm{v}' = \mathrm{v}[r]$, and $\mathrm{v}' \models \mathit{Inv}(l')$.
2. For $\delta \in \mathbb{R}_+$, $(l, \mathrm{v}) \xrightarrow{\delta} (l, \mathrm{v}')$ if $\mathrm{v}' = \mathrm{v} + \delta$, $\mathrm{v} \models \mathit{Inv}(l)$ and $\mathrm{v}' \models \mathit{Inv}(l)$.

A *run* ρ of A from a state (q_0, v_0) is a possibly infinite sequence $\rho = (q_0, \mathrm{v}_0) \xrightarrow{\delta_0} (q_0, \mathrm{v}_0 + \delta_0) \xrightarrow{a_1} (q_1, \mathrm{v}_1) \ldots \xrightarrow{a_n} (q_n, \mathrm{v}_n) \xrightarrow{\delta_n} \ldots$ where $\forall i$, $q_i \in L$, $a_i \in \Sigma$, $\delta_i \in \mathbb{R}_+$, $\mathrm{v}_i : X \to \mathbb{R}_+$, $\mathrm{v}_{i+1} = \mathrm{v}_i + \delta_i$ or $\mathrm{v}_{i+1} = \mathrm{v}_i[r_i]$ depending on the incoming transition and $q_i \xrightarrow[r_i]{[g_i]\ a_{i+1}} q_{i+1}$ $(\in E)$. The set of executions of A from a state s is denoted by $\mathit{Runs}_A(s)$. The set of runs of A is $\mathit{Runs}_A = \mathit{Runs}_A((l_0, \mathbf{0}))$. We say that a run is *$f$-faulty*, denoted $\mathit{faulty}(\rho, f)$, if $\exists i$ such that $a_i = f$. For a run ρ, let $\mathit{time}(\rho) = \sum_i \delta_i$, the sum of all delays in ρ. If ρ is an infinite run, then $\mathit{time}(\rho)$ is the limit of the sum (possibly ∞). We say that ρ is *non-Zeno* if $\mathit{time}(\rho) = \infty$ and *Zeno* otherwise.

The *trace* of a run ρ with respect to a set of observable actions Σ'_o, denoted $\mathit{trace}_{\Sigma'_o}(\rho)$, is the sequence $\delta_0 a_1 \delta_1 a_2 \ldots a_n \delta_n \ldots$ made only of time elapse and observable actions, i.e., $\forall i$, $a_i \in \Sigma'_o$ and $\delta_i \in \mathbb{R}_+$. If $\Sigma'_o = \Sigma_o$ (all the observables of a system), we obtain the usual trace definition.

A system is given by the parallel composition of the different timed automata it models. This means that the automata execute in parallel and synchronize on the common observable actions. We assume that the sets of clocks, silent actions and fault actions are mutually disjoint. This condition can be easily satisfied by renaming the common clocks or actions.

Definition 2 (Parallel composition). *Let $A_i = (L_i, l_0^i, X_i, \mathit{Inv}_i, \Sigma^i, E_i)$, $i \in \{1, 2\}$, be two TA such that $X_1 \cap X_2 = \emptyset$, $(\Sigma_u^s)^1 \cap (\Sigma_u^s)^2 = \emptyset$ and $(\Sigma_u^f)^1 \cap (\Sigma_u^f)^2 = \emptyset$. Their parallel composition denoted $A_1 \parallel A_2$ is the TA $(L, l_0, X, \mathit{Inv}, \Sigma, E)$ where*

- $L = L_1 \times L_2$,
- $l_0 = (l_0^1, l_0^2)$,
- $X = X_1 \cup X_2$,
- $\mathit{Inv} : L \to \Phi(X)$, $\mathit{Inv}(l_1, l_2) = \mathit{Inv}(l_1) \wedge \mathit{Inv}(l_2)$,
- $\Sigma = \Sigma_o^c \oplus \Sigma_o^u \oplus \Sigma_u^s \oplus \Sigma_u^f$ *with* $\Sigma_i^j = (\Sigma_i^j)^1 \cup (\Sigma_i^j)^2$, $(i, j) \in \{(o, c), (o, u), (u, s), (u, f)\}$,
- $E \subseteq L \times \Phi(X) \times \Sigma \times 2^X \times L$ *is the set of transitions given by*
  - $(l_1, l_2) \xrightarrow[r_1 \cup r_2]{[g_1 \wedge g_2]\ a} (l'_1, l'_2)$ *if* $a \in \Sigma^1 \cap \Sigma^2$, $l_1 \xrightarrow[r_1]{[g_1]\ a} l'_1$ *and* $l_2 \xrightarrow[r_2]{[g_2]\ a} l'_2$,
  - $(l_1, l_2) \xrightarrow[r_1]{[g_1]\ a} (l'_1, l_2)$ *if* $a \in \Sigma^1 \setminus \Sigma^2$ *and* $l_1 \xrightarrow[r_1]{[g_1]\ a} l'_1$, *and*
  - $(l_1, l_2) \xrightarrow[r_2]{[g_2]\ a} (l_1, l'_2)$ *if* $a \in \Sigma^2 \setminus \Sigma^1$ *and* $l_2 \xrightarrow[r_2]{[g_2]\ a} l'_2$.

Running example. Figure 1 presents a fragment of an autonomous system case study[1] that will be used as the running example throughout this paper. This

[1] The case study presented here is inspired from an autonomous robotics system. The original system contains more components, behavior and requirements.

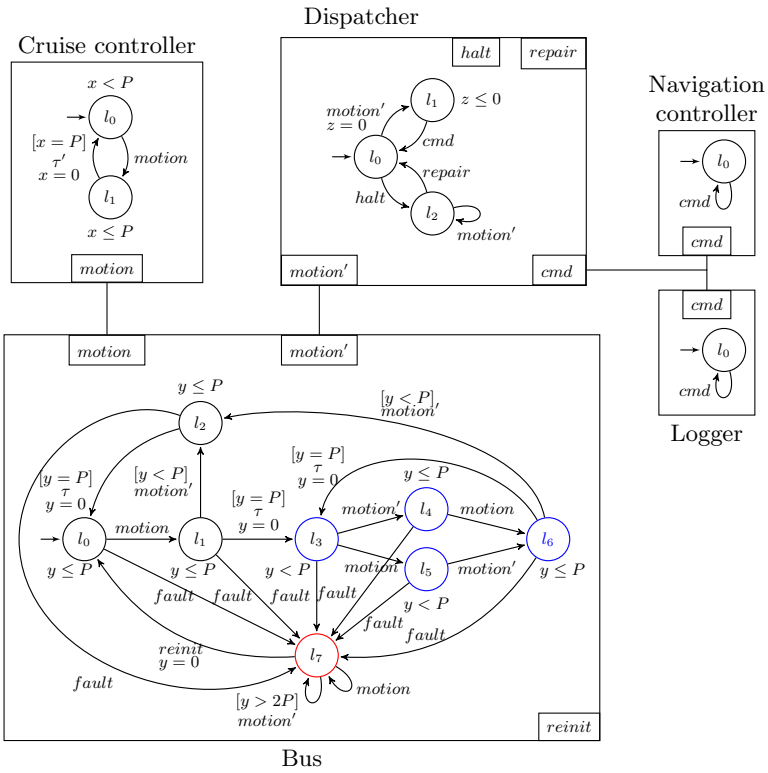

Fig. 1: An example of an autonomous cruise controller system with faults during communication.

system aims to transfer and dispatch motion commands from an automated cruise controller to the actual navigation system and a logger.

The system is modeled as a network of five communicating timed automata as follows. The *cruise controller* sends in every $[0, P)$ time interval the *motion* command to be executed.[2] This request travels through a *bus* to a *dispatcher*. The *dispatcher* sends the request to the *navigation controller* which is responsible for its mechanical execution. Additionally the request is stored in a *logger* for debugging purposes, possibly through replay.

The *bus* models a 2 element memory and has the following behavior. Once it receives a *motion* request and it is not busy, it can delay the request transfer up to P. In that case it waits for the next period of the *cruise controller* and "restarts" its behavior. This behavior describes the nominal mode of the bus and is depicted by the states in black in Figure 1 (from l_0 to l_2). However, the bus could be busy and the *motion* request is delayed for transfer at P or in the next

[2] For simplicity we abstract here the actual motion commands (possibly represented as multiple parameters of the *motion*, *motion*′, and *cmd* actions, respectively), and their mechanical execution.

period $[0, P)$. We then consider the bus to pass in a degraded mode depicted by the states in blue (from l_3 to l_6). During the transfer another request can be received. If both are handled before P, the bus has recovered and goes back to the nominal mode. Otherwise it stays in the degraded mode. While transferring the request, the bus can experience some hardware issues and fails denoted by the $fault$ action in the automaton. In this case the bus goes into the fail mode depicted by the l_7 red state, and in which the received requests are either delayed after $2P$, lost or multiplied.

The *dispatcher* assumes the nominal behavior of the *bus*: it receives requests steadily, within the $[0, 2P)$ period. If this requirement is not satisfied (due to a faulty behavior of the *bus*), the *dispatcher* must stop transferring requests (action *halt*). This means that the received motion commands are ignored until the network is reinitialized (action *reinit*) and the dispatcher is aware of it (action *repair*). The above description corresponds to the FDIR specification.

3 A Formal Approach for Designing FDIR Components

The design of FDIR components is a sub-process of the general system design, as illustrated in Figure 2. The methodology we propose includes several manual activities related to the design of the system that allow obtaining the inputs needed for the automated synthesis of FDIR components. These activities are suggested for system engineering by different standards, such as EECS standards [21] for space applications.

The main input of the methodology is the safety requirements. The first activity consists of building a system design from requirements and system description (i.e., what the system should do), which we call *requirements analysis*. The obtained design is usually made of two parts: the *nominal model* and the *fault model*. The nominal model defines the system architecture and its behavior in a "correct" environment (i.e., an environment that behaves accordingly to the assumptions). In this case, the nominal behavior should satisfy by default the (safety) requirements it is derived from. The fault model complements the nominal one by describing which faults components can manifest and what is the expected behavior after a fault occurrence. Usually the two models are obtained separately, since the fault model requires additional study of the fault specifications (e.g., of the hardware platform). Then, the two models are assembled into the *extended model* by merging techniques, which is used for FDIR design. For the sake of simplicity, we consider in the following that the output of the requirements analysis activity is the extended model.

Example. Figure 1 depicts the extended model of the case study, as the *bus* component models both nominal and faulty behavior. The nominal behavior consists of forwarding the *motion* request in the $[0, 2P)$ period – the nominal and degraded modes. The faulty behavior delays, loses or multiplies the *motion* requests after a $fault$ – the fail mode.

The second activity is the *Partitioning & Allocation*. Its aim is to associate the system requirements to the elements that must satisfy them, such as com-

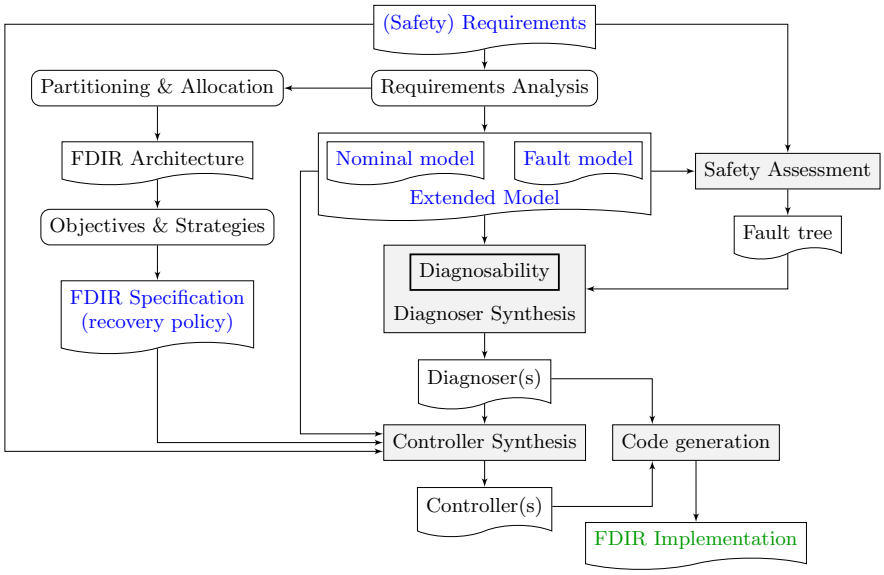

Fig. 2: Proposed formal approach for automated FDIR components design.

ponents, (sub-)systems or mission phases. As a result, the FDIR architecture is designed in relation to the system architecture and both nominal and fault-related requirements. The FDIR architecture can be centralized/distributed, monolithic/hierarchical or any combination. Please note that depending on the FDIR architecture, the automated design of FDIR component can become an undecidable problem, e.g., the decentralized partial observability control problem [30].
Example. For the case study in Figure 1, the requirement to satisfy is that *motion′* command is issued within one period in the best case and within two periods in the worst case. This deadline can be missed only in the case of a fault, when the *bus* becomes unresponsive and all messages are delayed after $2P$, and the *dispatcher* should not transfer any request. In consequence, the requirement is associated with the *bus* and *dispatcher* components. The FDIR component architecture we consider is a centralized flat one consisting of one diagnoser and one controller connected to the *bus* and *dispatcher* components.

The third and last activity is the *Objectives & Strategies*. From the FDIR architecture and the system requirements, an FDIR specification describing the recovery policy is derived. The recovery policy is defined at system level by *objectives* and at component level by *strategies*. Objectives are related to the system requirements for the fault model, i.e., required behavior in the presence of failures. Strategies usually contain the functional steps to be performed given the fault and the objective to achieve. The aim of a strategy is to bring the system back in a *good* (safe) state after faults, where the RAMS hold.
Example. In the running example the objective is to stop the *dispatcher* until the *bus* is reinitialized and the correctness of the *motion′* requests can be assumed.

The strategy to apply is the following: *halt* the dispatcher immediately after a fault is detected, *reinit* the *bus* and inform the *dispatcher* about the *bus* status (action *repair*).

Once all the inputs are clearly specified, the first automated step is *diagnoser(s) synthesis*. A *diagnoser* is a component that runs in parallel with the system and gives verdicts whether a fault has occurred or not yet. A diagnoser is synthesized for each fault (type) that can be detected. A fault f can be detected if the system satisfies the *diagnosability* condition: given a set of observable actions, there are no nominal and faulty executions (labeled with f) that have the same trace. Please note that possibly not all faults need to satisfy this condition, just as diagnosers do not have to be synthesized for all faults. Indeed, only a subset of the fault actions set could be relevant with respect to the safety requirements to ensure. These faults can be identified through *model-based safety assessment* techniques [14], such as building the *fault tree* (i.e., Boolean combinations of faults). Additionally, statistical model-checking could be applied to identify those faults most likely to happen. Describing how to perform safety assessment is outside the scope of this paper.

The second automated step is the *controller synthesis* which can be performed if the system's faults are diagnosable and the diagnosers have been synthesized. When a diagnoser detects a fault, an alarm is raised which triggers a controller. The controller is a component running in parallel with the system and implementing the recovery strategies specified for the fault (type). Its aim is to bring the system back to states/modes where the safety requirements hold. This step synthesizes a controller from the specified recovery strategies and with respect to the system and diagnoser(s) behaviors, thus ensuring the FDIR objectives.

Finally, the diagnosers and controllers are assembled into the FDIR component from which code is generated (in C++ for example). The generated code can be deployed and run online with the actual system implementation.

This approach is general enough to be applied for both untimed and timed systems, only the synthesis algorithms need to be adapted to the corresponding case. In the following we describe the algorithms for diagnoser and controller synthesis for timed systems with partial observability as formalized in Section 2, and we illustrate them on the example from Figure 1. These algorithms and the approach are currently under implementation in the BIP framework [5].

4 Formal Detection and Synthesis

A fault f can be detected by a diagnoser if the system is *f-Σ'_o-diagnosable*. Intuitively, a system is f-Σ'_o-diagnosable if there are no executions having the same trace with respect to a set of observables Σ'_o where one is labelled with the fault and the other not. With the notation from Section 2, the diagnosability condition is formalized as follows.

Definition 3 (f-Σ'_o-diagnosability). *Let S be a system represented as a TA, $f \in \Sigma_u^f$ and $\Sigma'_o \subseteq \Sigma_o^u$. S is f-Σ'_o-diagnosable if $\forall \rho \in Runs_S$ such that $faulty(\rho, f)$, $\nexists \rho' \in Runs_S$ such that $\neg faulty(\rho', q)$ and $trace_{\Sigma'_o}(\rho) = trace_{\Sigma'_o}(\rho')$.*

Please note in the definition above the *minimality condition* on the observables for fault detection: the set of observables is at most the set of uncontrollable observables of the system, i.e, $\Sigma'_o \subseteq \Sigma^u_o$. Additionally, the controllable observables should not be considered for the fault detection as those are actions commanded by the controller. If they would be taken into consideration for diagnosability, a circular behavioral dependency between the diagnoser and the controller could be created at runtime.

The algorithm for checking f-Σ'_o-diagnosability consists of the following steps:

1. Compute a copy A of S such that uncontrollable actions not in Σ'_o become silent actions and controllable actions and the corresponding transitions are removed from the TA: $A = (L^A, l_0^A, X^A, Inv^A, (\Sigma^u_o)^A \oplus (\Sigma^s_u)^A \oplus (\Sigma^f_u)^A, E^A)$ with $(\Sigma^u_o)^A = \Sigma'_o$, $(\Sigma^s_u)^A = (\Sigma^u_o \setminus \Sigma'_o) \oplus \Sigma^s_u$ and $E^A = E \setminus \{l \xrightarrow[r]{[g]\ a} l' | a \in \Sigma^c_o\}$.
2. Compute a copy B of A such that f is removed from the set of faults, clocks, silent and fault actions are renamed with respect to A, and all transitions labeled with f are removed from E. Formally, $B = (L^B, l_0^B, X^B, Inv^B, (\Sigma^u_o)^B \oplus (\Sigma^s_u)^B \oplus (\Sigma^f_u)^B, E^B)$ with X^B unique w.r.t. A, $(\Sigma^u_o)^B = \Sigma'_o$, $(\Sigma^s_u)^B = (\Sigma^u_o \setminus \Sigma'_o) \oplus \Sigma^s_u$ unique w.r.t. to A, $(\Sigma^f_u)^B = \Sigma^f_u \setminus \{f\}$ unique w.r.t. A, and $E^B = E^A \setminus \{l \xrightarrow[r]{[g]\ f} l'\}$.
3. Compute $A \parallel B$ and check that $\forall \rho \in Runs_{A \parallel B}$ such that $faulty(\rho, f)$, ρ is Zeno.

This algorithm is performed independently for every fault that might occur in the system (possibly only the relevant ones obtained through *safety assessment*).

Intuitively, the algorithm synchronizes two copies of the model from which the transitions labeled with controllable actions are removed. The copy A is the behavior with faults projected on the set of observables Σ'_o. The copy B is similar except the transitions labeled with the fault under study f are removed. The synchronization of A and B gives two types of executions: Zeno and non-Zeno. If a common execution labeled with f is non-Zeno, it means that the distinction of which execution was actually performed – with f from A or without f from B – cannot be made by the diagnoser. In contrast, if all runs ρ labeled with f are Zeno, the diagnoser is able to make the distinction after $time(\rho)$.

The definition and algorithm for checking diagnosability are similar to the ones in [29,18] for timed systems. The difference comes from the splitting of observable actions into controllable and uncontrollable, and the removal of controllable actions such that diagnosability does not depend on actions enforced by a controller as explained above. As these changes are linear in the number of actions considered, it follows from [29] that it runs in PSPACE.

The running example from Figure 1 has the following actions based on the components ports: $\Sigma^c_o = \{reinit, halt, repair\}$, $\Sigma^u_o = \{motion, motion', cmd\}$, $\Sigma^s_u = \{\tau, \tau'\}$ and $\Sigma^f_u = \{fault\}$. By taking $\Sigma'_o = \{motion'\}$, the system is $fault$-Σ'_o-diagnosable. In the construction above, all runs ρ labeled with $fault$ will reach the location (l_7, l_5) or (l_7, l_6) of the composition and $time(\rho) = 2P$. Therefore, such executions are Zeno and the system is diagnosable. The action $motion'$ gives in fact the minimal set of sensors for detecting a fault. Any

extended subset, e.g., monitoring also *motion*, preserves the diagnosability condition.

If a fault action satisfies the diagnosability condition, a diagnoser can be synthesized for its detection. Intuitively, a diagnoser monitors the observables on which diagnosability has been checked and raises an alarm when the states the system is in are marked as error. The diagnoser can be viewed, in general, as the TA obtained through determinization of the system under study and with respect to the specified observables Σ'_o and Σ^c_o. Determinization of TA is an undecidable problem [2,31], except for some classes [4,28,13,25]. In consequence, an algorithm for on-the-fly determinization of a copy of the system with marked faulty locations is generated. This algorithm is inspired from [29,18], and the differences are discussed below.

As mentioned, the diagnosis algorithm works on a copy of the system with marked faulty locations. Each location is associated with two bits: 0 if no fault has occurred and 1 otherwise. The transition relation is also duplicated: transitions labeled with an observable or silent action keep the bit of the source location, while transitions labeled with a fault change the bit to 1.[3] This modification is needed since the faults of the system are unobservable, and therefore the detection is based on the system state. We call this copy the *diagnosis model* and it is formalized as follows.

Definition 4 (Diagnosis model). *The diagnosis model S' for f-Σ'_o-diagnosable S is the TA $(L', (l_0, 0), X, Inv', \Sigma^c_o \oplus \Sigma'_o \oplus \{\tau\}, E')$ where $L' = L \times \{0,1\}$, $Inv'(l,n) = Inv(l)$ and E' is given by the relation:*

- $(l,n) \xrightarrow[r]{[g]\ a} (l',n)$ *for* $n \in \{0,1\}$ *if* $l \xrightarrow[r]{[g]\ a} l'$ *and* $a \in \Sigma^c_o \oplus \Sigma'_o$
- $(l,n) \xrightarrow[r]{[g]\ \tau} (l',n)$ *for* $n \in \{0,1\}$ *if* $l \xrightarrow[r]{[g]\ a} l'$ *and* $a \in (\Sigma^u_o \setminus \Sigma'_o) \oplus \Sigma^s_u \oplus (\Sigma^f_u \setminus \{f\})$
- $(l,n) \xrightarrow[r]{[g]\ \tau} (l',1)$ *for* $n \in \{0,1\}$ *if* $l \xrightarrow[r]{[g]\ f} l'$

The algorithm implemented by the diagnoser is given in Algorithm 1. We denote by W the set of current states for monitoring. Initially, this set consists of all states of the diagnosis model DM reachable in 0 time by firing only τ actions (i.e., former unobservable and observable actions on which the detection does not depend). Additionally, a Boolean variable b modeling whether a fault has been raised is set to false.

The diagnoser main implementation loop is given next. The current states are checked for being faulty, i.e., the bit 1 is set in the diagnosis model for all of them. If it is the case and no alarm has been raised so far (b is false), the latter is triggered. In both cases, the diagnoser keeps monitoring the system: (i) a clock x is set to 0 and (ii) an action a or the lapse of time to timeout is observed. For an observation a, x will contain the time elapse since the last match. In the case of a matched a, the diagnoser computes the next states of the system as follows:

[3] Please note that for simplicity of the diagnoser algorithm, all silent and fault actions are renamed as τ in Def. 4.

Algorithm 1: Timed diagnoser implementation loop.

Input: Diagnosis model $DM = (L \times \{0,1\}, (l_0, 0), X, \Sigma_o \cup \{\tau\}, E)$, $\mathsf{timeout} \in \mathbb{R}_+$, $x \notin X$ clock

$W \longleftarrow \{s \in L \times \mathbb{R}_+^X \mid \rho = (l_0, 0, \mathbf{0}) \xrightarrow{\tau*} s \wedge time(\rho) = 0\}$
while *true* **do**
  $b \longleftarrow false$
  while *true* **do**
    if $\forall s \in W.\ s = (l, 1, \mathrm{v})$ *and* $\neg b$ **then**
      raise alarm
      $b \longleftarrow true$
    end
    $x \longleftarrow 0$
    await (*action* a) or ($x = \mathsf{timeout}$)
    if a *is* restart **then**
      $W \longleftarrow \{(l, 0, \mathrm{v}) \mid \forall s \in W.\rho = s \xrightarrow{(\delta/\tau)*} (l, n, \mathrm{v}) \wedge time(\rho) = x\}$
      break
    else
      if *action* a **then**
        $W \longleftarrow \{s'' \mid \forall s \in W.\ \rho = s \xrightarrow{(\delta/\tau)*} s' \xrightarrow{a} s'' \wedge time(\rho) = x\}$
      else
        $W \longleftarrow \{s' \mid \forall s \in W.\ \rho = s \xrightarrow{(\delta/\tau)*} s' \wedge time(\rho) = \mathsf{timeout}\}$
      end
    end
  end
end

first it fires all internal actions τ such that the entire execution time takes x time units; then it fires the event a and updates the set of reached states. If the event a is the specific action restart of the controller, the bit is additionally set to 0 for all computed states. If no event is observed in a timeout period, the set of reachable states is again updated by firing all internals during the predefined timeout period.

Example. For the case study in Figure 1, the diagnoser has a timeout value of $2P$. Indeed at moment $2P$, the only state reachable is $(l_7, 1, 2P)$, at which the alarm action is raised. We give in Figure 3 a symbolic representation as TA for this diagnoser.

The difference between Algorithm 1 and the algorithm from [29,18] is mainly related to the FDIR setting and the controller component. While in [29,18] a valid diagnoser consistently outputs alarm once a fault is detected, our algorithm allows for a restart of the monitoring. This is due to the implementation of a controller which handles faulty behavior and brings the system into safe states. To ensure that the fault detection happens from states coherent with the actual system states after a restart, the diagnoser monitors also the controllable observable actions defined for the controller. Finally, for a more detailed discussion about the implementation of diagnosers, the reader is referred to [29].

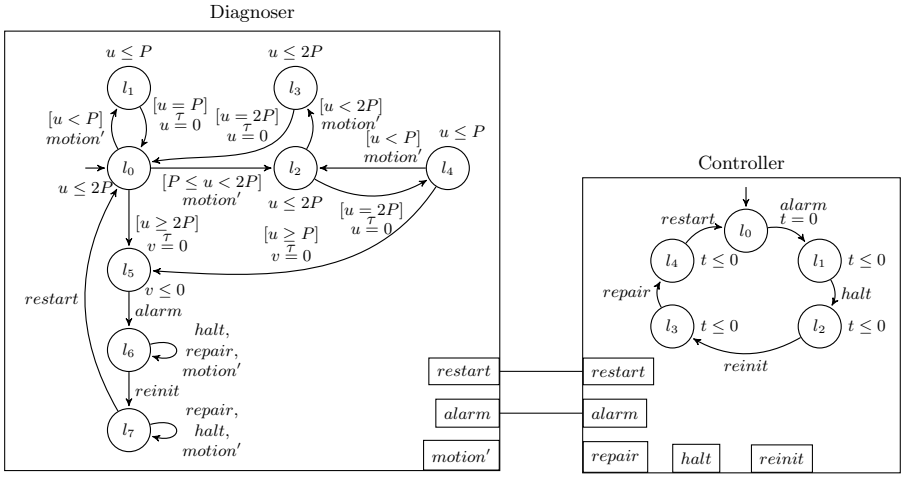

Fig. 3: Components of the synthesized FDIR component.

5 Implementing Reconfiguration Strategies

Once a fault is detected, a controller takes charge to bring the system back into safe states. This controller, if one exists, can be automatically synthesized just as the diagnoser. The construction of the controller is based on the system under study including diagnosers, FDIR specification (recovery strategies), and requirement(s) describing the safe state after a fault.

There are several works in the literature tackling controller synthesis algorithms from logical specifications (e.g. untimed automata built from LTL formulas [24]). The (safety) requirement is expressed in some logic and the algorithm works only on the system under study. This problem is known to be a hard one, and even undecidable in some cases [20,22]. In the FDIR context, we make use of the recovery strategies contained in the FDIR specification which define the functional steps to apply in case of a fault. An incomplete controller is manually built from these strategies, and refined and validated on the system.

We consider the incomplete controller to be modeled as a TA. This automaton is structurally refined by using the appropriate *alarm* and *restart* of the diagnoser, and adding a transition labeled with *restart* as last step of the modeled recovery strategy. This transformation is needed to be able to inform the corresponding diagnoser that the detected fault has been handled and to uphold the reactive aspect of these systems.

Example. On the running example, the recovery strategy describes that once the fault is detected, the *dispatcher* must stop transferring requests until the *bus* is reinitialized and it is aware of it. The functional sequence derived from this specification is *halt* followed by *reinit* and *repair*. This sequence is triggered by *alarm* from the diagnoser and informs its execution by action *restart*. The execution of the controller is enforced by invariants which deactivate time elapse. The corresponding TA is given in Figure 3.

The validity of a controller is checked with respect to the set of safe states to be reached modeled by the safety requirement. This can be achieved through model-checking the system including the FDIR component with respect to the safety requirement. As this is a reachability problem of the safe states, the validation effort of a controller is in PSPACE.

Example In our example, the safety requirement is that the *bus* has been reinitialized and the *dispatcher* is aware of it. The safe states projection on the two components consists of (l_0, l_0). With respect to this requirement the controller proposed in Figure 3 is valid.

Once the controller has been validated, code (e.g., in C++) can be generated. This code can be obtained independently for the diagnoser(s), controller(s) or the full-fledged FDIR component. Under the assumption of code generation correctness, the FDIR implementation can be deployed on the platform together with the actual system and its online expected behavior is met.

6 Related Work

How to build correct FDIR components from complete system specifications is a recent topic of interest in the literature [32,10]. FDIR components are usually obtained through *ad-hoc* processes and need extensive testing to ensure their correctness for mission and safety-critical applications. In this paper we propose the use of formal methods to tackle this issue and obtain correct-by-construction FDIR implementations.

Our approach is similar to the one in [10]. The main differences cover the domain of application and code generation feature. This paper considers real-time systems represented as timed automata, while in [10] systems are untimed and represented as symbolic transition systems. Therefore, the algorithms applied for synthesis are different. The focus of [10] is on the safety assessment [9] with timed failure propagation models, which are modeled by the user. In our approach, safety assessment is not mandatory as diagnosability can be checked for all fault types and diagnosers can be synthesized for all of them. However, safety assessment can be performed with respect to the given safety requirements based on automatically generated fault trees. The artifacts give information about the subset of faults that need to be detected and the minimal subset of sensors needed. Additionally, risk analysis can be performed along safety assessment using statistical model-checking (available in the BIP framework). The recovery strategies from [10] are modeled in a flavor of Linear Temporal Logic (LTL) [26], while we consider them specified as timed automata which is arguably more intuitive and expressive for modeling. Finally, our approach allows to generate code for the FDIR implementation, which is not considered in [10].

The two synthesis problems considered here – diagnoser and controller synthesis – have however been studied independently of the FDIR context. For example, [27] describes the diagnosability problem for one and multiple faults, and introduces the notion of fault type for untimed systems. [19] describes a framework for diagnoser synthesis in the untimed case and from LTL specifi-

cations. In [7,6], runtime verification for 3-value Timed LTL is used for fault detection.

In the context of timed diagnoser synthesis, the most related works to ours are the ones from [29,18,16,11]. As described in Section 4, the main differences consist of the representation of a system to accommodate FDIR components and the adaptation of the algorithms to this representation. More specifically, in our framework a system defines both controllable and uncontrollable observables, and diagnosability is checked on a subset of uncontrollable observables. The controllable actions are however monitored by the diagnoser, in order to ensure the correct restart of the detection once the recovery strategy is successfully applied. This feature of the diagnoser allows enforcing FDIR capabilities on reactive systems, different to [29] where a diagnoser is considered valid if it does not change the verdict after a detected fault.

The controller synthesis problem is studied in several works that use a more general formalism than ours. The usual approach is a game-based one: the problem is seen as a game between the environment (playing uncontrollable actions) and the controller [17]. In the untimed case, the general problem with specifications given as LTL formulas is well understood and decidable, but usually not tractable, even though some work has been done towards applicability of the algorithms [24]. An approach that is closer to ours in the untimed case, building a controller by adding transitions to an incomplete one under safety and liveness requirements, is considered in [3].

The timed case is much more involved, and the decidability of the problem heavily relies on parameters such as partial observability, access to (un)limited resources, and type of specifications [12,20,15]. For example, in [23] a controller is synthesized from a template by parameter instantiation, while [8,22] use Timed Computational Tree Logic [1] specifications.

7 Conclusion

In this paper we present an approach based on formal methods for the correct-by-construction design of FDIR components. This approach performs several manual steps for obtaining the inputs required: the system under study consisting of both the nominal and the faulty behavior, the recovery strategies and requirements to satisfy modeled as automata. The approach proceeds by synthesizing an FDIR component in two steps: (i) a diagnoser is generated for each diagnosable fault and (ii) a controller is produced for each recovery strategy by completion of its incomplete representation as automaton. The FDIR component is validated by model-checking techniques and code (e.g., in C++) is generated as FDIR implementation. This implementation can be deployed with the system for the online detection and enforcement of safety requirements.

The proposed approach can be applied for both untimed and timed systems. We define the notion of diagnosability in the FDIR context for timed systems and we propose algorithms for the automated generation of full-fledged timed FDIR

components. We illustrate the approach and the algorithms on an autonomous system case study with faults during communication.

The approach presented here is currently under implementation in the BIP framework and validation in two real-life case studies from our industrial partners. As future work we are interested in validating and comparing the synthesized FDIR implementation with respect to developer written ones, as means to quantify this approach with respect to standard FDIR coding ones. In order to optimize the synthesized FDIR implementations, we are interested to study model-based safety assessment with statistical model-checking techniques and in the context of stochastic faults.

On a more general note, we are interested in devising a pattern-based language for modeling recovery strategies, and a synthesis or learning algorithm for building the controller. This language could be extended and used for specifying data-/state-based safety requirements as inputs, besides the event-based ones considered in this paper. This would require the introduction of a dynamic observer as a filter for the FDIR component, to transform the data-/state-based property into an event-based one. Ideally, the previously mentioned algorithm will perform this step automatically. Finally, we are interested in considering more complex FDIR architectures as targets (e.g., distributed ones) and adapt the algorithms to such cases.

References

1. Alur, R., Courcoubetis, C., Dill, D.: Model-Checking in Dense Real-Time. Information and Computation 104(1), 2–34 (1993)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (Apr 1994)
3. Alur, R., Tripakis, S.: Automatic synthesis of distributed protocols. SIGACT News 48(1), 55–90 (2017)
4. Baier, C., Bertrand, N., Bouyer, P., Brihaye, T.: When are timed automata determinizable? In: ICALP 2009. pp. 43–54 (2009)
5. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: SEFM 2006. pp. 3–12 (2006)
6. Bauer, A., Leucker, M., Schallhart, C.: Model-based runtime analysis of distributed reactive systems. In: ASWEC 2006. pp. 243–252 (2006)
7. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: FSTTCS 2006. pp. 260–272 (2006)
8. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Beyond Liveness: Efficient Parameter Synthesis for Time Bounded Liveness. In: FORMATS 2005. pp. 81–94 (2005)
9. Bittner, B., Bozzano, M., Cavada, R., Cimatti, A., Gario, M., Griggio, A., Mattarei, C., Micheli, A., Zampedri, G.: The xSAP Safety Analysis Platform. In: TACAS 2016. pp. 533–539 (2016)
10. Bittner, B., Bozzano, M., Cimatti, A., Ferluc, R.D., Gario, M., Guiotto, A., Yushtein, Y.: An integrated process for FDIR design in aerospace. In: IMBSA 2014. pp. 82–95 (2014)
11. Bouyer, P., Chevalier, F., D'Souza, D.: Fault diagnosis using timed automata. In: FOSSACS 2005. pp. 219–233 (2005)

12. Bouyer, P., D'Souza, D., Madhusudan, P., Petit, A.: Timed control with partial observability. In: CAV 2003. pp. 180–192 (2003)
13. Bouyer, P., Jaziri, S., Markey, N.: On the determinization of timed systems. In: FORMATS 2017. pp. 25–41 (2017)
14. Bozzano, M., Villafiorita, A.: Design and Safety Assessment of Critical Systems. Auerbach Publications, Boston, MA, USA, 1st edn. (2010)
15. Cassez, F.: Efficient On-the-Fly Algorithms for Partially Observable Timed Games. In: FORMATS 2007. pp. 5–24 (2007)
16. Cassez, F.: A note on fault diagnosis algorithms. In: IEEE CDC 2009. pp. 6941–6946 (2009)
17. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: CONCUR 2005. pp. 66–80 (2005)
18. Cassez, F., Tripakis, S.: Fault Diagnosis of Timed Systems. In: Communicating Embedded Systems – Software and Design (Oct 2009)
19. Cimatti, A., Pecheur, C., Cavada, R.: Formal verification of diagnosability via symbolic model checking. In: IJCAI 2003. pp. 363–369 (2003)
20. D'Souza, D., Madhusudan, P.: Timed control synthesis for external specifications. In: STACS 2002. pp. 571–582 (2002)
21. European Cooperation for Space Standardization: Website, http://www.ecss.nl/
22. Faella, M., La Torre, S., Murano, A.: Dense real-time games. In: LICS 2002. pp. 167–176 (2002)
23. Finkbeiner, B., Peter, H.: Template-based controller synthesis for timed systems. In: TACAS 2012. pp. 392–406 (2012)
24. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless Compositional Synthesis. In: CAV 2006. pp. 31–44 (2006)
25. Lorber, F., Rosenmann, A., Nickovic, D., Aichernig, B.K.: Bounded determinization of timed automata with silent transitions. Real-Time Systems 53(3), 291–326 (2017)
26. Pnueli, A.: The temporal logic of programs. In: SFCS 1977. pp. 46–57 (1977)
27. Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., Teneketzis, D.: Failure diagnosis using discrete-event models. IEEE Trans. Contr. Sys. Techn. 4(2), 105–124 (1996)
28. Suman, P.V., Pandya, P.K.: Determinization and expressiveness of integer reset timed automata with silent transitions. In: LATA 2009. pp. 728–739 (2009)
29. Tripakis, S.: Fault diagnosis for timed automata. In: FTRTFT 2002. pp. 205–224 (2002)
30. Tripakis, S.: Undecidable problems of decentralized observation and control on regular languages. Inf. Process. Lett. 90(1), 21–28 (2004)
31. Tripakis, S.: Folk theorems on the determinization and minimization of timed automata. Inf. Process. Lett. 99(6), 222–226 (2006)
32. Wander, A., Forstner, R.: Innovative fault detection, isolation and recovery strategies on-board spacecraft: State of the art and research challenges. Deutscher Luft- und Raumfahrtkongress (2012)