Definition of Model-based diagnosis problems with Altarica

Yannick Pencolé[1] and **Elodie Chanthery**[1] and **Thierry Peynot**[2]
[1]LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
e-mail: {ypencole,echanthe}@laas.fr
[2]Queensland University of Technology (QUT) Brisbane QLD 4001, Australia
e-mail: t.peynot@qut.edu.au

Abstract

This paper presents a framework for modeling diagnosis problems based on a formal language called Altarica. The initial purpose of the language Altarica was to define a modeling language for safety analysis. This language has been developed as a collaboration between academics and industrial partners and is used in some industrial companies. The paper shows that the expressivity of this language, mixing event-based and state-based models, is sufficient to model classical model-based diagnosis problems (logic-based and event-based) and problems that combine state-based and event-based behaviors. This modeling framework, whose semantics is fully specified, is a promising framework to design richer diagnosis problems. As example, we introduce a robotic diagnosis problem that benefits from the full expressivity of the language.

1 Introduction

The research community of Model-Based Diagnosis (MBD) gathers a set of researchers that aim at solving diagnosis problems on many types of systems. Due to the large spectrum of investigated types of systems and applications, the diagnosis problems that are usually defined and solved rely on modeling languages that are more or less specific to the investigated problem. It follows that the solution proposed by the researchers cannot be compared or at least classified in a hierarchy of model-based diagnosis problems. Some researchers do not use any kind of human-friendly language and start developing their framework from fully specified mathematical tools, like first order logics (for static diagnosis problems), automata/Petri nets (for discrete-event systems) to name a few. The direct consequence is then the difficulty to actually promote diagnostic methodologies into industry as there is a gap the theoretical modeling framework and the way to model systems in practice.

In this paper, we adopt a new point of view.Our objective is to start from a modeling language that is actually used in industries and to investigate the type of model-based diagnosis problems that can be described with the help of this language. Our choice was to investigate the use of the Altarica language. This language has been specifically designed for modeling and analyzing safety problems especially in embedded systems. It has been designed by a consortium gathering academics and industrial partners and is used in industry (Dassault Aviation, Thales, Safran...). Our choice for the Altarica language also comes from the fact that some of our previous contributions focused on logic-based problems [Pencolé, 2014] and event-based problems [Pencolé and Cordier, 2005][Chanthery *et al.*, 2010] that can be all modeled in Altarica, as we will explain in this paper. We indeed show here that the Altarica language is expressive enough to model a set of diagnosis problems that are usually considered as different problems as they are not solved with the same diagnostic tools. We also propose the modeling of a specific problem from the robotics field that captures all the expressivity of an Altarica specification.

The paper is organized as follows. Firstly, the Altarica language is described and especially its semantics as a constraint automaton. Section 3 then proposes a way to model faults and observations with Altarica. Section 4 illustrates classical model-based diagnosis problems and the way they all can be modeled with Altarica. Section 5 introduces an original diagnosis problem from the robotics field and shows how the full expressivity of Altarica can be used to model it. The paper finally ends with a discussion and perspectives.

2 Altarica language

Altarica is a language used to describe safety critical systems that has become a European industrial standard for Model-Based Safety Assessment. It was initially designed within a consortium of academics and industrial partners two decades ago and results in the development of tools that are either academical tools (http://altarica.labri.fr/) or industrial tools like Cecilia OCAS from Dassault Aviation. This language aims at describing a system as a set of interacting components in a hierarchical way as a set of nodes and subnodes. We summarize here the description of the language and its semantics. Further details can be found in [Point, 2000; Point and Rauzy, 1999].

The following notations will be used in this paper. Let v denote a variable, the domain of the variable, denoted by $Dom(v)$, is the set of values that can be assigned to the variable v. A *valuation* of a variable v is the association of the variable v with one of its value $val \in Dom(v)$ and is denoted $(v = val)$. Let $\mathcal{V}$ denote a set of variables $Dom(\mathcal{V})$ will denote the product:

$$Dom(\mathcal{V}) = Dom(v_1) \times \cdots \times Dom(v_n)$$

where $\{v_1, \ldots, v_n\}, n \geq 1$ denotes the enumeration of the set $\mathcal{V}$ in a fixed order (like for instance the lexical order on the variable names). Any element e of $Dom(\mathcal{V})$ is an n-uple of values $(val_1, \ldots, val_n)$ that also obviously represents the

set of valuations $\{v_1 = val_1, \ldots, v_n = val_n\}$ or, in a logical form, the conjunction $\bigwedge_{i=1}^{n} v_i = val_i$. In the following, for the sake of notation simplicity, e will denote any of these three representations depending on the context.

2.1 Component in Altarica

A component is defined as a set of variables. There are two types of variables:

1. *State variable*: it is a variable that models the state of the component (keyword `state`).
2. *Flow variable*: it is a variable that represents a flow between the component and its environment (keyword `flow`).

The set of state variables is denoted V_S and the set of flow variables is denoted V_F.

Definition 1 (State)**.** *A* state *s of a component is a valuation of all its state variables,*

$$s \in Dom(V_S).$$

The state of a component is internal in the sense that the definition of a state does not take into account the valuation of the flow variables. Flow variables are seen as the interface of the component to its environment. In a given state of the component, it may be possible that one flow variable may have several valuations. At a given time, by definition, the flow and state variables have only one valuation, this set of valuations is called a configuration.

Definition 2 (Configuration)**.** *A* configuration *c is a valuation of all its state and flow variables,*

$$c \in Dom(V_S \cup V_F).$$

In the following, we might denote a configuration c as a couple (s, f) where $s \in Dom(V_S)$ and $f \in Dom(V_F)$. The relationship between flow and state variables is stated with the help of a set of *assertions* (keyword `assert`). An assertion is a logical condition between state and flow variables that must be true in any configuration of the component (in other words, any set of valuations of the state and flow variables that is inconsistent with at least one of the assertions is not an admissible configuration). The set of assertions is also called the *invariant* of the system. In a given state of the component, the environment may change and affect some flow variables of the component, so its configuration is changing, however this configuration change is constrained by the component's assertions that always must hold. Typically, the flow variables represent the inputs and outputs of a component to its environment. If a flow variable changes (interpreted here as an input variable), as a response, the component may also change other flow variables (interpreted here as output variables) to ensure the assertions are still true.

A component is an abstracted machine whose current configuration is modified by different phenomena:

1. change of the configuration due to an environmental change (change of a flow variable).
2. change of the internal state (and therefore the configuration) due to the occurrence of an internal event.

The notion of event is part of the Altarica language and has a meaning that is similar to the one in discrete event systems.

Definition 3 (event)**.** *An* event *is an instantaneous phenomenon that changes the state of a component.*

Events are explicitly defined below the keyword `event` in a node description and they are useful for the description of transitions (keyword `trans`). A transition is defined by a guard (or a precondition) that is a logical condition characterizing a subset of configurations. The transition is triggered only if the current configuration satisfies the precondition of the transition and if the event associated to the transition can occur. The consequence of the transitions is the immediate modification of a subset of state variables and possibly the immediate modification of the corresponding configuration to ensure the invariant of the component is satisfied.

Before recalling the semantics of an Altarica component, the abstracted syntax is presented.

Definition 4 (Altarica component)**.** *An* Altarica component[1] *is a 5-uple $\mathcal{C} = \langle V_S, V_F, E, I, M \rangle$ where*

- *V_S, V_F are two disjoint sets of variables, respectively called the set of state variables and the set of flow variables;*
- *$E = E_+ \cup \{\varepsilon\}$ is the set of events where $\varepsilon \notin E_+$;*
- *I is a first-order logic formula such that the free variables of I are in $V_S \cup V_F$;*
- *M is the set of macro-transitions (g, e, a) such that:*
  - *g is a first-order logic formula, called the guard, whose free variables are in $V_S \cup V_F$;*
  - *e is an event of E;*
  - *a is an application which maps any state variable $s \in V_S$ to a first-order logic term whose free variables are in $V_S \cup V_F$;*
  - *M contains the implicit transition $(\top, \varepsilon, a_\varepsilon)$ where $\forall v \in V_S, a_\varepsilon(v) = v$.*

The following example illustrates the definition.

```
node a_component
  flow
    pressure: {low,medium,high};
    switch:   {on,off};
  state
    working:  {yes, no};
  event
    start,stop;
  trans
    (working = no and pressure = low) |- start -> working:=yes;
    (working = yes) |- stop -> working:=no;
  assert
    if(working= no) then (pressure=low);
    if(working= yes and switch = on) then ( pressure=high);
    if(working= yes and switch = off) then ( pressure=medium);
edon
```

The flow variables are $V_F = \{\texttt{pressure, switch}\}$ and the state variable is $V_S = \{\texttt{working}\}$. The pressure variable represents here a physical interaction on the environment whereas the switch is a control of the component by the environment. There are two states in this component (working/not working). The state change is represented by two macro-transitions. The first macro-transition (represented as a 3-uple (g, e, a) in the previous definition) states that in every configuration where the component is

[1] Altarica can model events with priority and other subtleties. In this paper, we just omit these parts to avoid mathematical overload that does not have any impact on the topic discussed in this paper: priorities can indeed be included and then enrich the expressivity of the language, they have no other consequence on the matter discussed here.

working and the pressure is low (this is the guard g), the occurrence of a start event (this is the event e) will make the component to work (the application a returns the term `yes` when applied to variable state `working`). The second transition states that whatever the configuration is, the occurrence of a stop event on the working component makes it stop. Finally, the assertions model the behavior of the component regarding the flow variables. For instance, the first assertion states that when the component is not working (`working=no`), whatever the value of the switch is, the pressure is always low (`pressure=low`). With a component working and controlled by a *switch on* signal, the pressure is always high and if the switch is off, the pressure is medium. The formal semantics of a component behavior is described hereafter and relies on constrained automaton [Brleck and Rauzy, 1994].

Definition 5 (Constraint automaton)**.** *A* constraint automaton *is a 5-tuple $\mathcal{A} = \langle E, F, S, \pi, T\rangle$ where:*

- *$E = E_+ \cup \{\varepsilon\}$ is a finite set of events where ε is a special event that does not belong to E_+.*
- *F is the set of flow values.*
- *S is the set of automaton states.*
- *$\pi : S \to 2^F$ is a mapping that associates to any state of S the set of possible flow values in this state. Moreover, it is supposed that $\pi(s) \neq \emptyset$.*
- *$T \subseteq S \times F \times E \times S$ is the set of transitions such that:*
  - *$(s, f, e, s') \in T \Rightarrow f \in \pi(s)$;*
  - *$\forall s \in S, (s, f, \varepsilon, s) \in T$.*

In a constraint automaton, a state is associated to a set of possible flow values and the association between one state s and one of this flow value $f \in \pi(s)$ constitutes a configuration (s, f). The transition relation associates to one of this configuration, the occurrence of an event e and a target state s'. Finally, in this definition, it is also required that the event ε (which means "the non-occurrence of an event") can be triggered in any configuration (s, f) to stay in state s.

Definition 6 (Semantics)**.** *The semantics of an Altarica component $\mathcal{C} = \langle V_S, V_F, E, I, M\rangle$ is the constraint automaton $[\![\mathcal{C}]\!] = \langle E, F, S, \pi, T\rangle$ defined hereafter.*

- *$F = Dom(V_F)$.*
- *$S \subseteq Dom(V_S)$ is the set of states s that are admissible, that is: there exists f in F such that the configuration $c = (s, f)$ is such that the invariant I holds in c:*
$$c \models I.$$
- *The mapping $\pi : S \to 2^F$ is, for any $s \in S$,*
$$\pi(s) = \{f \in F | (s, f) \models I\};$$
- *$T \subseteq S \times F \times E \times S$ is the set of transitions $\{[\![(g, e, a)]\!]\}$ where $(g, e, a) \in M$ is a macro-transition and where $[\![(g, e, a)]\!]$ is the set of 4-uples (s, f, e, s') such that:*
  - *$s \in S$;*
  - *$f \in F$;*
  - *$s, f \models I$;*
  - *$s' = \{v_1 = val'_1, \ldots, v_n = val'_n\}$ where val'_i is the term $a(v_i)$ closed by the valuation s, f for all $i \in \{1, \ldots, n\}$.*

Figure 1 presents the constraint automaton of the previous example. As the previous example does not contain any initialization, any state of Figure 1 is an initial state.

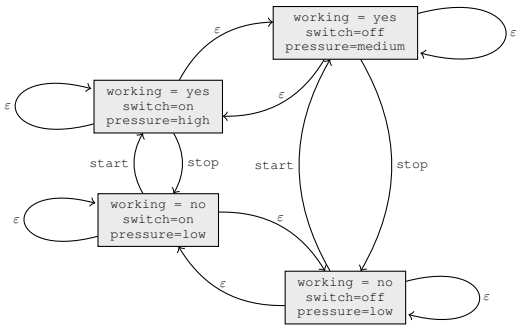

Figure 1: Constraint automaton describing the semantics of the example.

2.2 Compositional Modeling with Altarica

The basic piece of model in the Altarica language is a node. A node can model a component as presented in the previous section or a set of sub-nodes with the way they can interact with each other. Two sub-nodes can interact in two manners that are:

- *event based*: this type of interaction results from the synchronization of events that are defined in different sub-nodes. The syntax is:
```
sync
    <subnode1.e1,subnodes2.e2,...>;
```
This type of interaction is the classical way to synchronize components in discrete event systems. The event `e1` from the sub-node `subnode1` can only occur at the same time that the event `e2` from the sub-node `subnode2`.
- *signal based*: the signal based interaction is implemented as an equality constraint between two flow variables of two different sub-nodes, the syntax is:
```
assert
    subnode1.v1 = subnode1.v2;
```
This constraint ensures that at any time both variables `v1` from `subnode1` and `v2` from `subnode2` have the same value which means that if the sub-node `subnode1` assigns a new value to `v1`, due its internal behavior, it instantaneously sends a signal to `subnode2` to assign the same value to `v2`. Changing the value of `v2` may also have some internal and instantaneous effect in `subnode2`. Note that the interaction might be bidirectional (`subnode2` might affect `subnode1` the same manner `subnode1` affects `subnode2`).

It follows from this that the semantics of *any* node $\mathcal{N}$ can actually be represented as a constrained automaton. The definition of this automaton is done by induction. If $\mathcal{N}$ does not contain any sub-node, its semantics is the constrained automaton presented in the previous section. Now suppose that $\mathcal{N}$ contains a non-empty set of sub-nodes $\mathcal{N}_1, \ldots, \mathcal{N}_n$, we can associate to the node $\mathcal{N}_i, i \in \{1, \ldots, n\}$ its semantics $[\![\mathcal{C}_i]\!] = \langle E_i, F_i, S_i, \pi_i, T_i\rangle$. The node $\mathcal{N}$ might also have some state variables V_S (state variables that are declared within the node $\mathcal{N}$). It might also have some flow variables V_F (flow variables declared within the node $\mathcal{N}$), events $E = E^+ \cup \{\varepsilon\}$, invariant I (a logical formula asserting a property between the state variables of V_S, the flow variables of V_F and the flow variables V_F^i declared in any

sub-node $\mathcal{N}_i, i \in \{1, \ldots, n\}$), and finally macro-transitions M as any Altarica component. This part of the node can be described as any component $\langle V_S, V_F \cup \bigcup_{i=1}^{n} V_F^i, E, I, M \rangle$ and is called the *controller* of the node $\mathcal{N}$: it has its own semantics $[\![\mathcal{C}_0]\!] = \langle E_0, F_0, S_0, \pi_0, T_0 \rangle$, as any Altarica component. Regarding the set of events of the node $\mathcal{N}$, they might be involved in synchronizations or not. A global event of node $\mathcal{N}$ is a $n+1$ vector of events $\langle e_0, \ldots, e_n \rangle$ where e_i are in E_i that matches the synchronization rules defined by the `sync` keyword. The empty event of node $\mathcal{N}$ is $\langle \varepsilon, \ldots, \varepsilon \rangle$ that is also denoted as ε by convention. If an event $e_i \in E_i$ is not synchronized with any other events, its corresponding global event is $\langle \varepsilon, \ldots, e_i, \ldots, \varepsilon \rangle$. The set of global events of node $\mathcal{N}$ is denoted E.

Definition 7 (Semantics of a node)**.** *Let $\mathcal{N} = \langle V_S, V_F, E, I, M, \mathcal{N}_1, \ldots, \mathcal{N}_n \rangle$ a node, its semantics is the constrained automaton $[\![\mathcal{C}]\!] = \langle E, F, S, \pi, T \rangle$ such that:*

- $F = Dom(V_F)$*;*
- $S = \{s \in S_0 \times S_1 \times \cdots \times S_n | \pi(s) \neq \varnothing\}$.
- $\pi(\langle s_0, \ldots, s_n \rangle) = \{f \in F | \exists \langle f, f_1, \ldots, f_n \rangle \in \pi_0(s_0), \forall i \in \{1, \ldots, n\}, f_i \in \pi_i(s_i)\}$.
- $T \subseteq S \times F \times E \times S$ *such that:* $\langle (s_0, \ldots, s_n), f, \langle e_0, \ldots, e_n \rangle, (s'_0, \ldots, s'_n) \rangle$ *if there exists* $f_0 = (f, f_1, \ldots, f_n) \in \pi_0(s_0)$ *and for any* $i = \{0, \ldots, n\}$, $f_i \in \pi_i(s_i)$ *and* $(s_i, f_i, e_i, s'_i) \in T_i$.

With the semantics of a node, the language Altarica provides a formal characterization of a dynamical system as a constraint automaton $[\![\mathcal{C}]\!] = \langle E, F, S, \pi, T \rangle$. The model is designed as a hierarchy of nodes that provides a clear distinction between the structural part of the system (how the nodes interact with each others), its functional part (how a node reacts to a given set of inputs) and its behavioral parts (how a node behaves when events occur in the system). In this sense, Altarica provides a way to model a system as defined in [Chittaro *et al.*, 1993].

3 Modeling of observations and faults

3.1 Observed system

The first task required by any diagnostic engine is to *observe* the system. To do so, the model first need to express what is observable and what is not observable. The notion of observations in the sense that is commonly used in the diagnosis communities is not included in the language. However, Altarica provides the possibility to add tags in the declaration of events and variables. Two types of observation can then be considered.

1. *Observable variables*: a variable might be a flow variable or a state variable. If a flow variable is observable, it represents that a signal (either an input signal or an output signal), a synchronous interaction between nodes is observable. A state variable usually represents a property of the internal state of a node. This property can be also observable. Syntactically speaking, we propose that an observable variable v is described in Altarica by adding a tag `observable` to its declaration (whatever it is a state or a flow variable):

   ```
   v: observable;
   ```

2. *Observable events*: as stated in [Point, 2000] and recalled in Definition 3, an event is considered as a *instantaneous phenomenon* that modifies the state of the component. The language makes no distinction about the nature of the event, that can be coming from an environmental stimulus or can be an internal event. Such an event might be observable or not observable. Syntactically speaking, we also propose that an observable event e is described in Altarica by adding a tag `observable` to the declaration of the event:

   ```
   event e: observable;
   ```

Adding tags to events and variables is the syntactical way to express how a system is observed in the Altarica model. Now we need to define the formal semantics of this observation model.

Definition 8 (Observable valuation mask)**.** *The observable variable mask obs is a function that maps any valuation to a logical formula,*

- $obs(v = val) = (v = val)$ *if v is observable;*
- $obs(v = val) = \top$ *(true) if v is not observable.*

This observable valuation mask can be extended to any set of valuations:

$$obs(\{v_1 = val_1, \ldots, v_n = val_n\}) =$$

$$obs(v_1 = val_1) \wedge \cdots \wedge obs(v_n = val_n).$$

Each configuration of a node is partially observable through an observable valuation mask.

Definition 9 (Observable Part of a Configuration)**.** *The observable part of a configuration c is $obs(c)$.*

The observable mask only applies on states, configurations. Now, let formally define observable events.

Definition 10 (Observable event mask)**.** *The observable event mask eobs is a function that maps any event to an event:*

- $eobs(e) = e$ *if e is observable;*
- $eobs(e) = \varepsilon$ *if e is not observable.*

Note that $eobs$ also applies to ε and $eobs(\varepsilon) = \varepsilon$. The observable event mask can also be extended to a set of synchronized events $\langle e_1, \ldots, e_n \rangle$ as follows:

$$eobs(\langle e_1, \ldots, e_n \rangle) = \langle eobs(e_1), \ldots, eobs(e_n) \rangle.$$

From the definitions of mask here above, it is now possible to define the observable part of any transition $t \in T$ of the constrained automaton associated to any type of node.

Definition 11 (Observable Part of a Transition)**.** *The observable part of a transition $\langle s, f, e, s' \rangle$ in a constrained automaton $[\![\mathcal{C}]\!]$ associated to a node $\mathcal{N}$ is $\langle obs(s), obs(f), eobs(e), obs(s') \rangle$.*

Now that the way to model what is observable in an Altarica specification is fully defined, we can define the type of observations that can be stated in an Altarica diagnosis problem. An *observation* of the system for a given diagnosis problem should consist of:

1. *observed events* from E_O on one hand;
2. *observable variable changes* on the other.

The first type of observation is similar to the one that is modeled in a fault diagnosis problem. The second one is related to the fact that a variable is observed. Suppose that the model has an observable variable v (flow or state variable), by stating that a variable is observable we mean that, at anytime, it is possible to know its value so especially at the time when the value of the variable changes.

Definition 12. *An observation OBS of the system modeled by a node $\mathcal{N}$ is a non-empty sequence of couples $(e, c) \in (E_O, \mathcal{L}(V_O))$ with V_O the set of observable variables and $\mathcal{L}(V_O)$ the logical language on the valuations of V_O.*

An observable OBS is a non-empty sequence, as we might be able to observe something at the time the system starts operating, this first observation is actually the initialization of observable variables that is represented as $(\varepsilon, \bigwedge_{v \in V_O} v = val_{init})$. It might also be possible that no observable variable are available, in this case, the observation is *nothing* represented as $(\varepsilon, \top)$. The system may evolve and then generates the change of values for a set of variables V_O^1, it is represented within OBS by $(\varepsilon, \bigwedge_{v \in V_O} v = val_{init}).(\varepsilon, \bigwedge_{v \in V_O^1} v = val_1)$ and so on. During its evolution, the system might also generate an observable event o (or a synchronized event like for instance $\langle \varepsilon, \varepsilon, o_1, \varepsilon, o_2, \dots \rangle$) and some instantaneous value changes l, that is represented as (o, l). If the occurrence of the event o does affect any observable variable, it is represented as $(o, \top)$.

3.2 How to model faults, failures in an Altarica specification

A diagnosis problem aims at determining *what went wrong* given a set of observations. In a Model-Based diagnosis framework, the '*what went wrong*' part is specified in the model but mostly depends on the objectives of the diagnosis that is performed on the system. We might be interested in *malfunctioning behaviors* by determining the functional modes of the components and identify the ones that are tagged as *failing modes*. We might be interested in identifying *fault events* that cause the problems. Within an Altarica specification, we actually propose the user to model any kind of these diagnostic objectives.

- A *failure* is usually defined as a phenomenon where a system does not operate properly, that is it does not proceed as expected with the given inputs. If such a system is specified as an Altarica node, the failure then occurs only if the system is in a *failure state*. To model these failure states, we propose to declare some state variables as failure state variables, as follows:
  ```
  state   var: bool : failure
  ```
- A *fault* is usually a phenomenon that is the cause of a system's failure. Within an Altarica specification, we propose to model it as a *fault event*, as follows:
  ```
  event   e : fault
  ```

Depending on the diagnosis objectives and the underlying system, we might model only failure states (see Section 4.1 for an example) or only fault events (see Section 4.2) or even both (see Section 5). By using failure state variables, we actually implicitly specify *failure modes*. Let $V_{FS} \subseteq V_S$ be the set of state variables that are declared as failure variable.

Definition 13 (Failure mode)**.** *A failure mode m is a set of valuations for each variable of V_{FS},*

$$m \in Dom(V_{FS}).$$

A state s has a failure mode m if:

$$s \models m.$$

4 Definition of Model-based diagnosis problems

The purpose of this section is to illustrate the expressivity of the Altarica language by presenting within the same modeling framework, two classical model-based diagnosis problems that are usually studied independently.

4.1 Static Diagnosis Problem

This problem, commonly called the polybox problem, consists of a digital circuit (the circuit of [Davis, 1984]) made of a set of multipliers $M_1, M2, M3$ and adders A_1, A_2. The initial problem is a static problem in the sense that the observations OBS represent one timepoint and the state of the system is supposed to be permanent.

```
node multiplier
     flow
        x,y : RANGE : in;
        p : RANGE : out;
     state
        ab: bool: failure;
     assert
        not ab => (p = (x*y));
edon

node adder
     flow
        x,y : RANGE : in;
        s   : RANGE : out;
     state
        ab: bool: failure;
     assert
        not ab => (s = (x+y));
edon

node circuit
flow
     a,b,c,d,e : RANGE : in,observable;
     f,g : RANGE : out,observable;

sub
  A1,A2: adder;
  M1,M2,M3: multiplier;
assert
  M1.p = A1.x; M2.p = A1.y; M2.p = A2.x;
  M3.p = A2.y; a=M1.x; b=M2.x; c=M1.y;
  c=M3.x; d=M2.y; e=M3.y; f=A1.s; g=A2.s;
edon
```

Figure 2: Description of the circuit in Altarica.

Figure 2 fully presents the model of the circuit. The system is actually modeled with the node `circuit` that consists of 5 sub-nodes. Each sub-node has a type (that is either `adder` or `multiplier`). Flow variables at this level are the inputs (`in`) and the outputs (`out`) of the system and they are all observable. The `assert` section defines the structural model of the circuit (the links). For each sub-node, there is one state variable `ab` that is boolean and that just asserts whether the sub-node is abnormal `ab = true` or not `ab = false`. The functional behavior of each sub-node is finally described as an assertion.

The diagnosis problem, introduced in [Reiter, 1987], was introduced as follows. Given a system description SD (in first order logic), a set of observations OBS (a valuation of the observable variables) and a set of components $COMPS$, find a diagnosis Δ as a subset of $COMPS$ such that $SD \wedge OBS \wedge \bigwedge_{c \in \Delta} Ab(c) \wedge \bigwedge_{c \notin \Delta} \neg Ab(c)$ is satisfiable. In the Altarica world, the problem is defined like this. Given the Altarica model M as the one of figure 2, given

$OBS_M = (\varepsilon, OBS)$ find in $[\![C_M]\!]$ a configuration that satisfies the observation OBS_M and extract its failure mode. In this case, a failure mode is a valuation of all the `ab` state variables so it corresponds to an initial diagnosis Δ.

4.2 Discrete Event Diagnosis Problem

The second problem that is presented here is the classical fault diagnosis problem on discrete event system illustrated by the HVAC system [Sampath *et al.*, 1995].

```
node valve
state V: [1,4];
event        stuck_closed1, stuck_closed2: fault;
       stuck_open1, stuck_open2: fault;
       open, close;
trans
  V=1 |- close -> V:=1; V=1 |- open -> V:=2;
  V=1 |- stuck_open1 -> V:=4; V=1 |- stuck_closed1 -> V:=3;
  V=2 |- open -> V:=2; V=2 |- close -> V:=1;
  V=2 |- stuck_open2 -> V:=4; V=2 |- stuck_closed2 -> V:=3;
  V=3 |- open -> V:=3;V=3 |- close -> V:=3; V=4 |- open -> V:=4;
  V=4 |- close -> V:=4;
init V:=1;
edon

node pump
state P: [1,4];
event start, stop;
      failed_on1, failed_on2, failed_off1, failed_off2: fault;
trans
  P=1 |- stop -> P:=1; P=1 |- start -> P:=2;
  P=1 |- failed_on1 -> P:=4; P=1 |- failed_off1 -> P:=3;
  P=2 |- start -> P:=2;        P=2 |- stop -> P:=1;
  P=2 |- failed_on2 -> P:=4; P=2 |- failed_off2 -> P:=3;
  P=3 |- start -> P:=3;        P=3 |- stop -> P:=3;
  P=4 |- start -> P:=4;        P=4 |- stop -> P:=4;
init P:=1;
edon

node controller
state C: [1,4];
event open, close, start, stop: observable;
trans
 C=1 |- open  -> C:=2; C=2 |- start -> C:=3;
 C=3 |- stop  -> C:=4; C=4 |- close -> C:=1;
 init C := 1;
edon

node hvac
sub v: valve; p: pump; c: controller;
sync
    <c.open,v.open>; <c.close,v.close>;
    <c.start,p.start>; <c.stop,p.stop>;
edon
```

Figure 3: Description of the HVAC system in Altarica.

The HVAC system is modeled as a `hvac` node. This node is composed of three sub-nodes: a `valve`, a `pump` and a `controller`. In this example, the structural model is represented as event synchronizations only described in the `hvac` node. Each sub-node is modeled as a set of transitions that are triggered by events that modify the internal state of each sub-node. Here, there is no observation of any state, only the events of the controller are observable. The fault information is carried on a subset of events. The keyword `init` defines the initial state of each sub-node, so here it also defines the initial state of the system.

The model presented on Figure 3 is purely event-based, the underlying constrained automaton of this system is equivalent to the global model G defined in [Sampath *et al.*, 1995]. It is indeed, by construction, a pure Mealy machine without any ε-transitions since there is no flow variables, no assertions. The first diagnosis problem that can be defined on the HVAC system is: given OBS a sequence of observable events, find a subset F of fault events such that there exists in the model G a trajectory from the initial state, containing as fault events exactly the set F, that satisfies OBS. Back to the Altarica world, the problem is defined as follows. The sequence of observations OBS (for instance $open, start, stop, close, start$) becomes OBS_M:

$$(\langle c.open, \varepsilon\rangle, \top), (\langle c.start, \varepsilon\rangle, \top),$$

$$(\langle c.stop, \varepsilon\rangle, \top), (\langle c.close, \varepsilon\rangle, \top), (\langle c.start, \varepsilon\rangle, \top).$$

The diagnosis problem then becomes: find a subset F of events declared as `fault` such that there exists in the constrained automaton a trajectory from the initial configuration that satisfies OBS_M, trajectory that contains, as fault events, the set F exactly. The second problem that can be defined is the one that is solved by the diagnoser approach of [Sampath *et al.*, 1995]. Instead of just computing a set of faults F, the problem is defined as determining the set of couples (x, F) where x is global state of the system. In the model of Altarica, it means that it is required to keep track of the current failure mode: in this particular case, a failure mode is exactly one state. To keep track of the failure mode, it suffices to add the keyword `failure` to any state variable of the model. Then to finally get (x, F), it suffices to extract from the last configuration of the trajectory the failure mode to get x.

5 State/Event-based Diagnosis Problem: a UGV diagnosis problem

In the previous section, we show that it is possible to specify with the Altarica language two classical problems:

1. the first one is *state-based*, the observations provide a partial information about the underlying state of the system;
2. the second one is *event-based*, the observations are a sequence of events and the purpose is to determine which fault events have occurred.

The main strength of Altarica is its ability to provide into one fully specified specification language event-based behaviors (Mealy machines) and state-based behaviors (Moore machines). Our proposal is now to take advantage of this expressivity to model *state/event-based* diagnosis problems and thus refine the type of information that is provided within the model. To illustrate our objective, we investigated a case study: a diagnosis problem in Robotics.

We consider a model of an unmanned ground vehicle (UGV) capable of autonomous navigation in challenging environmental conditions, such as in the presence of fog, smoke or airborne dust. In order to achieve safe autonomous navigation, the UGV needs to be equipped with a perception system capable of performing obstacle detection. This task is critical since any detection error could lead to catastrophic consequences, hence the motivation to perform diagnosis on this perception system. Figure 4 shows a high-level view of our proposed model of a perception system. In the model in Fig. 4, the sensors observe aspects of the environment from their mounting position on the UGV, providing sensor data. These sensor data, e.g. 3D points, are transformed into a common world frame by a function `RawDataToWorld`, thanks to calibration parameters (e.g. the position and orientation of the sensor in a frame related to the vehicle) and the full localization of the vehicle in the world. Once this operation is performed, the data can be fused into a common representation, which is then interpreted. An example of this interpretation for obstacle detection is making the distinction between obstacle and free space. The

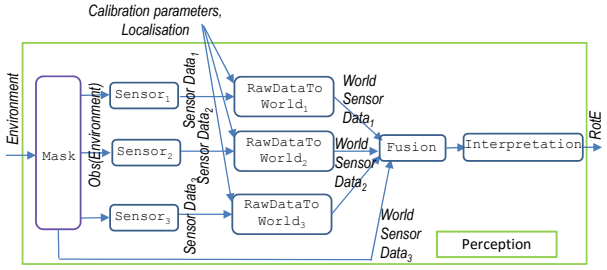

Figure 4: UGV perception component overview

output of the perception model is then a representation of the environment (*RotE* in Fig. 4). Each function in that model, namely $\texttt{sensor}_i$, $\texttt{RawDataToWorld}_i$, `Fusion` and `Interpretation` contains a submodel. Figure 5 shows one of these submodels, for $\texttt{sensor}_1$.

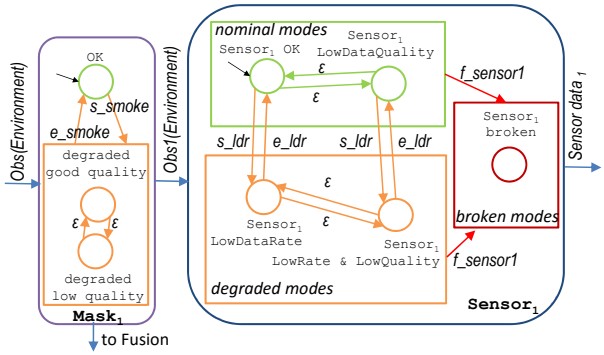

Figure 5: Model for one sensor

Note that our perception system has multiple sensors (3 in our case). This is because it has been shown that using multiple sensors, and more particularly multiple sensing modalities (e.g. LIDARs, RADARs, and visual/IR cameras), is key to perform resilient perception in challenging environmental conditions [Peynot *et al.*, 2009], as the UGV perception system can benefit from the complementarity between the distinct multiple sensing modalities. For example, mm-wave RADARs can see through smoke or airbone dust while LIDARs often cannot, which means returns from LIDARs are often misinterpreted as obstacles, however, LIDARs usually are much more accurate than RADARs [Castro and Peynot, 2012]. These differences of perception make the problem of fusing and interpreting data acquired by multiple sensing modalities particularly difficult [Gerardo-Castro *et al.*, 2014], as they can generate many failures of obstacle detection. To address this challenge, the originality of our modeling example is that it includes a model of the environment mask so that different observation configurations are possible. For example, if smoke occurs, the observation mask of the sensors that are sensitive to smoke (in the previous example, the LIDAR) can be degraded and consequently the diagnosis process can take the environment mask as responsible for a degraded mode.

Figure 6 fully shows the model of one sensor of the proposed UGV model. The sensor is modeled as a `sensor` node. This node is composed of two sub-nodes: the observation mask $\texttt{mask}_1$ of the sensor and $\texttt{sensor}_1$. Each elementary node contains several behavioral modes and the mode changes by the occurrence of events. For instance, in $\texttt{sensor}_1$, there are three operating modes (see Figure 5) and three events (`s_ldr`,`e_ldr`,`f_sensor1`) that affect the current mode of the sensor. Among these events, one is considered as a fault event (`f_sensor1`) that is not recoverable and the others (namely start/end of low data rate: `s_ldr`,`e_ldr`) are not considered as faults (as `s_ldr` is actually recoverable). This part of the model is the event-based part, it handles the way that instantaneous phenomena like `s_ldr` or `f_sensor1` affect the internal state of the system. The state-based part of the model is composed of the flow variables and the set of assertions. It is used to model the fact that in the presence of smoke the sensor is not able to produce high quality data *even if its operating mode is nominal*. Indeed, the cause of the low-quality data might not be due to a problem in the sensor (a low data rate issue) but due to the presence of smoke in the environment. This is modeled by asserting that the presence of smoke in the mask leads to a low quality input of the sensor. In this example, the model contains failure modes (one diagnostic objective is to estimate the current failing mode based on which UGV reconfigurations can be automatically planned during the mission) and fault events (a second diagnostic objective is to estimate the health of the UGV for maintenance after the mission).

```
node mask_1
state
   mode: {ok, degraded};
flow
   quality: {good,low};
event
   s_smoke, e_smoke: observable;
trans
    mode=ok |- s_smoke -> mode:=degraded;
    mode=degraded |- e_smoke -> mode:=ok;
init
    mode:=ok;
assert
   (mode=ok) => (quality=good);
edon

node sensor_1
flow
   input: {good,low};
   output: {empty,degraded,ok};
state
   mode: {nominal, degraded, broken}:failure;
   rate: {good,low}:failure;
event
    s_ldr, e_ldr: observable;
    f_sensor1: fault;
trans
    mode !=broken |- f_sensor1 -> mode:=broken;
    mode = nominal and rate = good |- s_ldr -> mode:=degraded, rate:=low;
    mode = degraded and rate=low   |- e_ldr -> mode:=nominal, rate:=good;
assert
   ((mode=nominal) and (input=good)) => (output=ok);
   ((mode=nominal) and (input=low)) => (output=degraded);
   ((mode=degraded) and (rate=low)) => (output=degraded);
   (mode=broken) => (output=empty);

init
    mode:=nominal;
    rate:=good;
edon

node sensor
flow
    data: {empty,degraded,ok};
sub
    m: mask_1;
    s: sensor_1;
assert
    m.quality = s.input;
    s.output = data;
edon
```

Figure 6: Part of the UGV system in Altarica.

6 Discussion

This aim of the paper is to present a specification language that has the great advantage to have a precise semantics as a constrained automaton. Within this modeling framework, we show that it is possible to model classical diagnosis problems that are rather different in the literature and these models do not use the full expressivity power of Altarica. Moreover, we investigated a case study from the robotics field where we need to use all the expressity power of Altarica to express a more complex diagnosis problem that is a combination of a state-based problem and an event-based problem.

This paper did not discuss the way to effectively solve diagnosis problems within Altarica. Solving such generic problems with one diagnosis engine is our perspective. There already exist tools, like ARC, that are able to perform model analysis on Altarica specifications. Among these analyses, it is in particular possible to use commands like *sequences* and *cuts*: a cut can be seen as a function that extracts from the Altarica model an abstracted set of trajectories that match a specification. It is a way to explore the model from initial configurations to final configurations and extract relevant pieces of information. This is particularly this type of methods that [Kuntz *et al.*, 2011] use to design diagnostic rules from an Altarica specification. This type of approach attempts to compile the pieces of information that are necessary to run a diagnostic engine. What we propose here as Altarica specifications of diagnosis problems is actually more generic than what is proposed in [Kuntz *et al.*, 2011] and cannot be compiled as simple diagnostic rules. One naive way to solve a complete diagnostic problem written in Altarica would be to update the diagnoser approach [Sampath *et al.*, 1995] to the underlying constrained automaton of the Altarica specification. This is indeed theoretically possible if the domain of the variables are finite however, we believe that it is not the right way to go due to the space complexity issue. We think that the best way to go would be to benefit from a close integration of tools that actually perform state-based diagnosis like [Feldman *et al.*, 2010; Pencolé, 2014; Shchekotykhin *et al.*, 2015] with tools that perform event-based diagnosis [Lamperti and Zanella, 2002; Pencolé and Cordier, 2005; Grastien and Anbulagan, 2013]. Investigating a diagnosis problem also require the diagnosability analysis of a problem. ARC is actually a model-checker that has been specifically developed to model-check Altarica specifications so it could be use to check diagnosability questions as well as NuSMV [Bozzanoa *et al.*, 2014].

References

[Bozzanoa *et al.*, 2014] Marco Bozzanoa, Alessandro Cimatti, Oleg Lisagorb, Cristian Mattareia, Sergio Movera, Marco Roveria, and Stefano Tonettaa. Safety assessment of altarica models via symbolic model checking. *Science of Computer Programming*, 98(4):464–483, 2014.

[Brleck and Rauzy, 1994] S. Brleck and A. Rauzy. Synchronization of constrained transitions systems. In *First International Symposium on Parallel Symbolic Computation*, 1994.

[Castro and Peynot, 2012] M.P.G. Castro and T. Peynot. Laser-to-radar sensing redundancy for resilient perception in adverse environmental conditions. In *ARAA Australasian Conference on Robotics and Automation (ACRA)*, Sydney, Australia, December 2012.

[Chanthery *et al.*, 2010] Elodie Chanthery, Yannick Pencolé, and Nicolas Bussac. An ao*-like algorithm implementation for active diagnosis. In *10th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, pages 378–385, Sapporo, Japan, 8 2010.

[Chittaro *et al.*, 1993] L. Chittaro, G. Guida, C. Tasso, and E. Toppano. Functional and teleological knowledge in the multimodeling approach for reasoning about physical systems: a case study in diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6):1718 – 1751, 1993.

[Davis, 1984] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

[Feldman *et al.*, 2010] Alexander Feldman, Gregory Provan, and Arjan van Gemund. Approximate model-based diagnosis using greedy stochastic search. *Journal of Artificial Intelligence Research*, 38:371–413, 2010.

[Gerardo-Castro *et al.*, 2014] Marcos P. Gerardo-Castro, Thierry Peynot, Fabio Ramos, and Robert Fitch. Robust multiple-sensing-modality data fusion using gaussian process implicit surfaces. In *IEEE International Conference on Information Fusion*, Salamanca, Spain, July 2014.

[Grastien and Anbulagan, 2013] Alban Grastien and Anbu Anbulagan. Diagnosis of discrete event systems using satisfiability algorithms: a theoretical and empirical study. *IEEE Transactions on Automatic Control (TAC)*, 58(12):3070–3083, 2013.

[Kuntz *et al.*, 2011] Fabien Kuntz, Stéphanie Gaudan, Christian Sanino, Éric Laurent, Alain Griffault, and Gérard Point. Model-based diagnosis for avionics systems using minimal cuts. In *22nd International Workshop on Principles of Diagnosis*, 2011.

[Lamperti and Zanella, 2002] Gianfranco Lamperti and Marina Zanella. Diagnosis of discrete-event systems from uncertain temporal observations. *Artificial Intelligence*, 137:91–163, 2002.

[Pencolé and Cordier, 2005] Yannick Pencolé and Marie-Odile Cordier. A formal framework for the decentralised diagnosis of large scale discrete event systems and its application to telecommunication networks. *Artificial Intelligence*, 164(2):121–170, 5 2005.

[Pencolé, 2014] Yannick Pencolé. Dito: a csp-based diagnostic engine. In *21st European Conference on Artificial Intelligence*, pages 699–704, Prague, Czech Republic, 8 2014.

[Peynot *et al.*, 2009] Thierry Peynot, James Underwood, and Steven Scheding. Towards reliable perception for unmanned ground vehicles in challenging conditions. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1170–1176, Saint Louis, MO, USA, October 2009.

[Point and Rauzy, 1999] G. Point and A. Rauzy. Altarica - constraint automata as a description language. *European Journal on Automation*, 33:1033–1052, 1999.

[Point, 2000] G. Point. *AltaRica: Contribution à l'unification des méthodes formelles et de la Sûreté de fonctionnement*. PhD thesis, Université Bordeaux I, 2000.

[Reiter, 1987] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 4 1987.

[Sampath *et al.*, 1995] Meera Sampath, Raja Sengupta, Stéphane Lafortune, Kasim Sinnamohideen, and Demosthenis Teneketzis. Diagnosability of discrete-event systems. *Transactions on Automatic Control*, 40(9):1555–1575, 9 1995.

[Shchekotykhin *et al.*, 2015] Kostyantyn Shchekotykhin, Dietmar Jannach, and Thomas Schmitz. A divide-and-conquer-method for computing multiple conflicts for diagnosis. In *Proceedings of the 26th International Workshop on Principles of Diagnosis (DX-2015)*, pages 3–10, 2015.