SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2007; **37**:365–415
Published online 24 October 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.773

A diagnostic environment for automaton networks

S. Cerutti[1], G. Lamperti[2,*,†], M. Scaroni[3], M. Zanella[2] and D. Zanni[4]

[1]*I.M.I. s.r.l., Bergamo, Italy*
[2]*Dipartimento di Elettronica per l'Automazione, Università degli Studi di Brescia, Italy*
[3]*ONION S.p.A., Brescia, Italy*
[4]*S4WIN s.r.l., Brescia, Italy*

SUMMARY

Automated diagnosis of communicating-automaton networks (CANs) is a complex task, which is typically faced by model-based reasoning, where the behavior of the network is reconstructed based on its observation. This task may take advantage of knowledge-compilation techniques, where a large amount of reasoning is anticipated off-line (when the diagnostic process is not active), by simulating the behavior of the network and by constructing suitable data structures embedding diagnostic information. This (general-purpose) compiled knowledge is exploited on-line (when the diagnostic process becomes active), so as to generate the solution to the problem. Additional reusable (special-purpose) compiled knowledge is generated on-line when solving new problems. A software environment for the diagnosis of CANs has been developed in the C programming language with the support of the PostgreSQL relational database management system, under the Linux operating system. It supports the modeling and preprocessing of CANs as well as the solution of diagnostic problems, including on-line knowledge compilation. The environment has been tested through a variety of experiments. Results are encouraging and provide a valuable feedback for further work. Copyright © 2006 John Wiley & Sons, Ltd.

Received 24 August 2005; Revised 16 June 2006; Accepted 18 June 2006

KEY WORDS: diagnosis; model-based reasoning; communicating automata; knowledge compilation; reuse

1. INTRODUCTION

Automated diagnosis of dynamic (or time-varying) physical systems [1,2] is a complex task. Communicating-automaton networks (CANs) are a formalism that can be adopted for the (distributed) specification of dynamic systems, where each component is represented by a finite-state machine that can communicate with other components through links. Since, at some level of abstraction,

*Correspondence to: Gianfranco Lamperti, Dipartimento di Elettronica per l'Automazione, Via Branze 38, 25123 Brescia, Italy.
†E-mail: lamperti@ing.unibs.it

most real-world systems can be viewed as CANs and reasoning about such networks is easier than about continuous systems, from the middle of the 1990s the task of diagnosis of CANs has been receiving an increasing amount of interest from both the artificial intelligence [3–8] and the automatic control communities [9–16].

Diagnosing a network means computing its candidate diagnoses, each of which is a set of faults that explains the observation collected during the network operation. In the general case, the specific faults of a network cannot be inferred without finding out what has happened to the system [17]. In this way, the system evolutions complying with the observation, called the histories [18], situation histories or narratives [19], paths [20], or trajectories [21], become a product of the diagnostic reasoning. Determining the system evolutions is computationally expensive (see [22] for the difficulties of the diagnoser approach [9,10], or the worst-case computational complexity analysis in [18], or the discussion in [23]). This is why most approaches exploit a trade-off between off-line and on-line computation: some kind of knowledge, implicit in the models of the structure and behavior of the network, is compiled off-line in order to speed up on-line processing.

While the approaches by other authors deal with networks wherein the communication between automata is supported by synchronous links, the CANs taken into account here and in previous works by the present authors constitute an adaptation of communicating finite-state machines [24,25], these being networks of finite-state machines that asynchronously exchange messages with each other through FIFO links. In particular, this paper presents a diagnostic environment for asynchronous CANs endowed with *finite* FIFO links (other policies of link management are possible, but outside the scope of this work). The foundation of the diagnostic method is the approach outlined in [26], but, in addition, it integrates the outcomes of the present authors' more recent research about knowledge compilation and similarity-based reasoning techniques [27]. The environment was developed in the C language under the Linux operating system.

In the remainder of the paper, Section 2 illustrates the notion of a CAN adopted in the implemented environment. Section 3 introduces the concept of a diagnostic problem inherent to such a CAN. Section 4 states the requirements for the diagnostic environment and outlines the architecture of the software system. Section 5 points out the major choices pertinent to the design of the knowledge base. Section 6 deals with the specification language for describing CANs and diagnostic problems inherent to them. Section 7 defines some relevant algorithms for solving diagnostic problems by exploiting knowledge reuse and similarity-based reasoning. Section 8 details some aspects of the implementation. Section 9 draws the experience in using the diagnostic environment and provides the obtained results. Section 10 discusses the limitations of the diagnostic environment. Finally, Section 11 draws conclusions and envisages future work.

2. AUTOMATON NETWORKS

The diagnostic environment presented in this paper focuses on networks of communicating-automata [7,18,26,28] and is a partial implementation of the techniques introduced in [27]. A CAN is the abstraction of a physical system, such as a protection apparatus or a telecommunication network. Each automaton represents the behavior of a basic component, such as a breaker or a protection. Although not all physical systems can be conveniently modeled as CANs, a significant variety of them may be viewed, at some level of abstraction, as such.

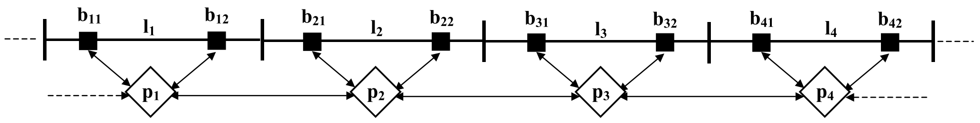

Figure 1. Power transmission lines.

Automata may communicate through special channels called *links*. Considered stand-alone, an automaton is a finite-state machine equipped with input and output terminals. The automaton makes a transition (changes its state) upon the arrival of an event at an *input terminal*. A transition consumes the input event and possibly generates further events at its *output terminals*.

Since each link connects an output terminal of an automaton with an input terminal of another automaton, the output event generated by the transition is queued into the link for further consumption by the receiver. Therefore, each automaton is both a consumer and a producer of events. Each link is characterized by a *capacity*, the maximum number of storable events. When the link is full, an attempt to insert a new event results in the loss of the event.

Each automaton is implicitly equipped with a particular input terminal for connection with the external world, called the *standard input*, denoted *In*. This terminal makes the automaton sensitive to external events that are relevant to the behavior of the network. For example, considering a power transmission network, a typical external event is the occurrence of a short circuit on a line.

A *subnetwork* $\aleph'$ of a network $\aleph$ is a subgraph of $\aleph$ involving a subset of the automata of $\aleph$ along with all of the links connecting such automata in $\aleph$. In fact, a subnetwork is a network itself. A network is said to be *closed* if all terminals (standard input aside) are connected to a link of the network, otherwise the network is said to be *open*. In an open network, terminals that are not connected are said to be *dangling*.

We say that two networks $\aleph$ and $\aleph'$ are *isomorphic*, denoted $\aleph \doteq \aleph'$, when:

(1) they involve the same number of automata and links;
(2) for each automaton $A \in \aleph$ there exists an automaton $A' \in \aleph'$ such that A and A' are the same finite-state machine, and *vice versa* (we denote the relationship between A and A' as $A \between A'$);
(3) for each link $L \in \aleph$ there exists a link $L' \in \aleph'$ such that L and L' have the same capacity, and *vice versa* (we denote the relationship between L and L' as $L \between L'$);
(4) if $L \between L'$, where $L \in \aleph$ connects output terminal O_1 of A_1 to input terminal I_2 of A_2, and $L' \in \aleph'$ connects output terminal O'_1 of A'_1 to input terminal I'_2 of A'_2, then $A_1 \between A'_1$ and $A_2 \between A'_2$.

Example 1. We consider a sample application domain inherent to power transmission lines. Each line is protected by two breakers that are commanded by a protection. The protection is designed to detect dangerous conditions. Typically, if a short circuit affects the line, the protection is expected to trip the two breakers to open. In a simplified view, the system is represented by a series of lines, each one associated with a protection, as displayed in Figure 1, where lines $l_1 \cdots l_4$ are protected by protections $p_1 \cdots p_4$. For instance, p_2 controls l_2 by operating breakers b_{21} and b_{22}. In normal (correct) behavior, both breakers are expected to open when tripped by the protection. However, the protection system may exhibit an abnormal (faulty) behavior, for example, one breaker

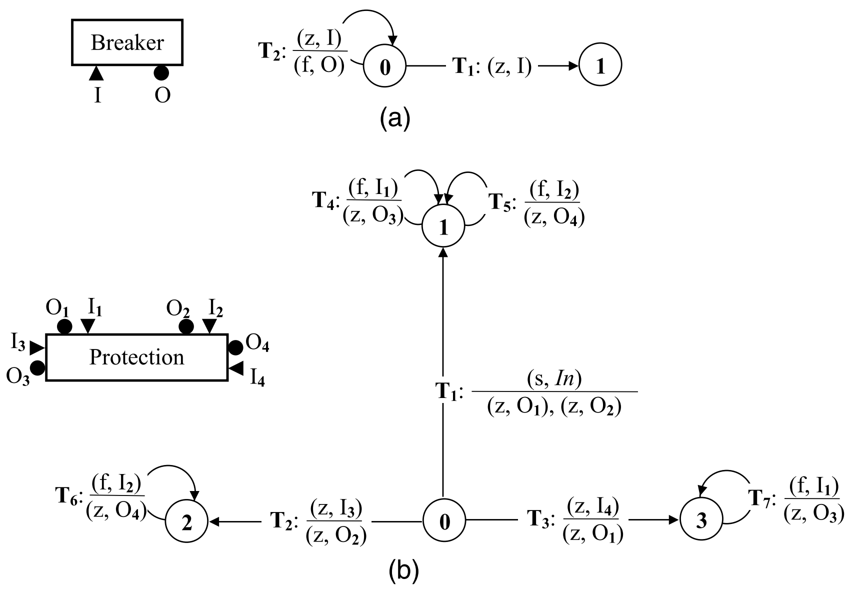

Figure 2. Communicating automata.

or both may not open when required. In such a case, each faulty breaker informs the protection about its own misbehavior. Then, the protection sends a request of recovery actions to the neighboring protections, which will operate their own breakers appropriately. For example, if p_2 operates b_{21} and b_{22} and the latter is faulty, then p_2 will send a signal to p_3, which is supposed to command b_{32}. A recovery action may be faulty on its turn. The protection system is designed to propagate the recovery request until the tripped breaker opens correctly. When the protection system is reacting, a subset of the occurring events are visible to the operator in a control room who is in charge of monitoring the behavior of the power transmission system and, possibly, of issuing explicit commands so as to minimize the extent of the isolated subsystem. The localization of the short circuit and the identification of the faulty breakers may be impractical in real contexts, especially when the isolation spans several lines and the operator is required to make a decision within stringent time constraints. On the one hand, there is the problem of observability: the observable labels generated during the reaction of the protection system are generally incomplete and uncertain in nature. On the other hand, whatever the 'quality' of the observation, it is impractical for the operator to reason on the observations so as to make consistent hypotheses on the behavior of the system and, eventually, to establish the shorted line and the faulty breakers.

Example 2. Displayed in Figure 2 are the automata *Breaker* (Figure 2(a)) and *Protection* (Figure 2(b)), relevant to the protection system outlined in Figure 1. Each automaton is associated with a box indicating its input and output terminals. Considering the breaker, the input terminal I is depicted

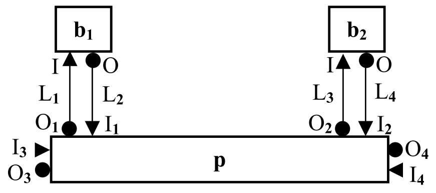

Figure 3. *Net* communicating-automaton network.

as a triangle, while the output terminal O is represented as a bullet. The automaton incorporates two states, marked by 0 (closed) and 1 (open), respectively, and two transitions, T_1 and T_2, represented as arrows between states. The graphical representation of a transition mirrors its formal specification: a transition T from state S to state S', which is triggered by an event e at input terminal I, and generates events $e_1, \ldots, e_k$ at output terminals $O_1, \ldots, O_k$, respectively, is denoted by

$$T = S \xrightarrow[(e_1,O_1),\ldots,(e_k,O_k)]{(e,I)} S'$$

When the breaker is closed (state 0), either transition T_1 or T_2 is non-deterministically triggered by event z on input terminal I. T_1 moves the breaker to state 1 (open) without generating any output event. T_2, instead, keeps the state of the breaker unchanged (closed), whilst generating event f at output terminal O. Intuitively, this is an abnormal transition, as the breaker is supposed to open when triggered, which is not the case for T_2.

The protection embodies four input terminals, $I_1 \cdots I_4$, and four output terminals, $O_1 \cdots O_4$. Terminals O_1 and I_1 are meant for connection with the breaker on the left of the device, while terminals O_2 and I_2 are for the communication with the breaker on the right. I_3 and O_3 allow the protection to exchange events with the neighboring protection on the left. The same applies for I_4 and O_4, which are a means to communicate with the adjacent protection on the right. The corresponding automaton involves four states, marked by $0 \cdots 3$, and seven transitions, $T_1 \cdots T_7$. State 0 stands for the normal condition.

The occurrence of a short circuit on the protected line is signaled by event s at the standard input *In*, which triggers transition T_1. Such a transition moves the protection to state 1 by generating event z at both output terminals O_1 and O_2, thus commanding the two breakers to open. In state 1, the protection may receive event f either at terminal I_1 or I_2, meaning that the relevant breaker failed to open. This triggers either transition T_4 or T_5, respectively, each of which generates event z at output terminals O_3 and O_4, respectively. This reaction complies with the recovery actions described in Example 1, where the failure of a breaker requires the intervention of a breaker driven by the adjacent protection.

When a protection receives a request of recovery from a neighboring protection, it performs a transition from state 0 to either 2 or 3, depending on whether the request comes from the left (T_2) or from the right (T_3), respectively. So, input event (z, I_3) causes T_2 to generate (z, O_2), that is, a command to the breaker on the right. In state 2, since even this breaker may in turn be faulty, the occurrence of event (f, I_2) triggers transition T_6 that, similarly to T_5, propagates the recovery request to the right-hand side protection. A symmetric behavior is defined in state 3.

The topology of a network *Net* is depicted in Figure 3, which integrates the three elements protecting a generic device, namely protection p and breakers b_1 and b_2. The protection is connected with the breakers by means of links $L_1 \cdots L_4$. We assume that all links can store at most one event (capacity = 1). Note the dangling terminals I_3, O_3, I_4 and O_4. This means that transitions of protection p may be triggered by events on I_3 and I_4 as well as by the short circuit external event. System *Net* is the abstraction of a subpart of the protection system of a power system. Larger subparts of the system may be assembled by connecting instances of *Net* by means of links among protections.

A network $\aleph$ may be either *idle* or *reacting*. When idle, all links in $\aleph$ are empty and no transition is performed. Upon the occurrence of an event either at the standard input or a dangling terminal, $\aleph$ becomes reacting by performing a transition of one of its automata, which possibly generates other events toward other automata, thereby continuing reacting until all links are empty anew. Thus, the activity of $\aleph$ is an intermittent sequence of inactivity and reaction conditions. When reacting, $\aleph$ may evolve only within a confined space that is constrained by its topology and the nature of its automata and links. Each configuration of this space is a network state, specifically, a pair $\sigma = (\mathbb{A}, \mathbb{L})$, where $\mathbb{A}$ is a record of the states of the automata in $\aleph$, while $\mathbb{L}$ is a record of the states of the links in $\aleph$. Let $\aleph_0$ be the initial (idle) state of $\aleph$. Upon the occurrence of an external event, $\aleph$ becomes reacting, thereby making a series of network transitions, namely a *history* of $\aleph$. Each network transition is the transition of one automaton in $\aleph$. The whole set of possible evolutions of $\aleph$ from $\aleph_0$ are specified by a so-called *behavior space*, written $Bhv(\aleph, \aleph_0)$, which is the automaton describing the behavior of $\aleph$ as a whole. Specifically, the set of states encompasses all of the network states reachable from $\aleph_0$ via network transitions. The set of final states includes the network states where all links are empty. Each transition between two network states σ_1 and σ_2 is denoted by

$$\sigma_1 \xrightarrow{T} \sigma_2$$

where T is the transition of an automaton in $\aleph$. Intuitively, based on $\sigma_1 = (\mathbb{A}_1, \mathbb{L}_1)$ and T, the new state $\sigma_2 = (\mathbb{A}_2, \mathbb{L}_2)$ can be computed by the following steps:

(1) initialize $\sigma_2 := \sigma_1$;
(2) update $\mathbb{A}_2$ with the state reached by T;
(3) if the input event triggering T is internal to $\aleph$, then remove it from $\mathbb{L}_2$;
(4) append into $\mathbb{L}_2$ all of the output events generated by T (unless the relevant links are full).

More precisely, a history relevant to the initial state $\aleph_0$ is a path in the behavior space represented by the sequence of transitions marking the edges of the path, which starts at $\aleph_0$ and ends at a final state. The whole set of histories encompassed by a behavior space $\mathcal{B}$ is denoted by $\|\mathcal{B}\|$. We implicitly generalize the notion of extension to all sorts of graphs for which initial and final states are defined.

Example 3. In Figure 4 the behavior space relevant to network *Net* depicted in Figure 3 is shown, where the initial (quiescent) state is $Net_0 = (\mathbb{A}_0, \mathbb{L}_0)$, where $\mathbb{A}_0 = (0, 0, 0)$. Quiescent nodes are double circled. In each node, the record $\mathbb{A}$ of the component states for b_1, p and b_2 is on the top, while the record $\mathbb{L}$ of queues of events within links $L_1 \cdots L_4$ is on the bottom. Since at most one event is stored in each link, the state of the link can be expressed by either the label of the event or a dash, the latter denoting the empty link. Nodes are numbered. For instance, in node 7 both breakers are closed (state 0 of the breaker model), while the protection has commanded the breakers to open (state 1 of the protection model). In addition, links L_1 and L_4 are empty; instead, L_2 incorporates event f

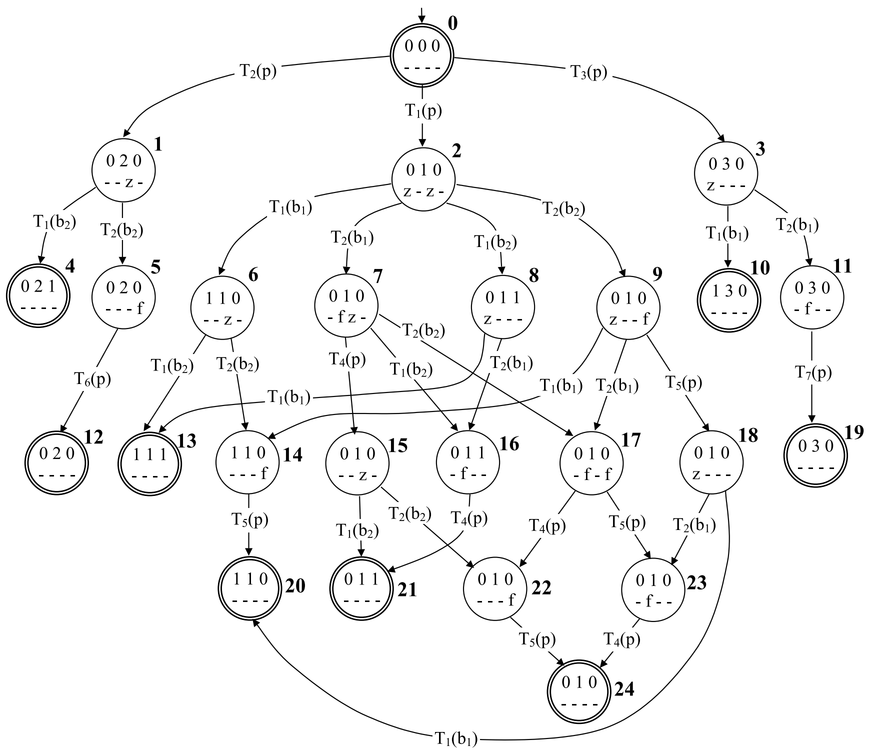

Figure 4. Behavior space of *Net*.

(meaning that b_1 has failed to open) while L_3 contains event z, meaning that b_2 has not yet reacted to the protection command. Each edge is marked by a transition followed by the name of the relevant component. Note how *Net* becomes reacting upon the occurrence of an external event, either from the standard input (triggering transition $T_1(p)$) or from a dangling terminal (triggering either $T_2(p)$ or $T_3(p)$). Each (possibly empty) path from the initial state to a final state is a history of *Net*. A history $h(Net)$ is identified by the sequence of labels (component transitions) marking the edges on such a path, as for instance, $h(Net) = \langle T_1(p), T_2(b_1), T_1(b_2), T_4(p)\rangle$. The transitions in $h(Net)$ lead us to the following scenario: (1) a short circuit occurs on the device protected by *Net*, hence protection p commands both breakers b_1 and b_2 to open; (2) breaker b_1 fails to open; (3) breaker b_2 opens correctly; (4) protection p asks the neighboring protection on the left to perform a recovery action.

A peculiarity of the behavior space in Figure 4 is acyclicity, so that the number of histories is finite. As pointed out above, however, a behavior space may include cycles, encompassing an unbounded number of histories.

3. DIAGNOSTIC PROBLEM

A diagnostic problem concerns a complete reaction of the network. This means that we assume the expiration of the network reaction before starting the diagnostic task, namely, we perform *a posteriori diagnosis*. Solving a diagnostic problem amounts to finding out possible misbehaviors in the network reaction. A diagnostic problem is expressed by means of some information about the network to be diagnosed and some clues on the network reaction. Formally, a diagnostic problem $\wp$ for a network $\aleph$ is a record

$$\wp(\aleph) = (\aleph_0, \mathcal{V}, \mathcal{O}, \mathcal{R}, \mathcal{K})$$

where:

- $\aleph_0$ is the *initial state* of $\aleph$, when the reaction started;
- $\mathcal{V}$ is the *viewer*, with specific visibility properties;
- $\mathcal{O}$ is the *observation* of the system generated during the reaction, based on viewer $\mathcal{V}$;
- $\mathcal{R}$ is the *ruler*, which establishes what behavior is to be considered faulty;
- $\mathcal{K}$ is the *knowledge* about the system, including at least the structural and behavioral description of $\aleph$ and, possibly, additional compiled knowledge.

If $\mathcal{K}$ does not encompass any compiled knowledge, $\wp(\aleph)$ is a *crude problem*, otherwise it is a *compiled problem*. The solution of a compiled problem is bound to be more efficient than that of a crude problem, since part of the model-based reasoning necessary for solving the problem is somehow codified in $\mathcal{K}$.

3.1. Viewer

A viewer $\mathcal{V}$ establishes at diagnostic-problem time those transitions that are somewhat visible, as well as the specific observable label for each of them. $\mathcal{V}$ is defined by associating each transition T of each automaton in $\aleph$ with an observable label ℓ. A special *null label* ε is defined. If $\ell \neq \varepsilon$, then the transition is *observable* else T is *silent*. This reflects on the observability of $\aleph$: if T is observable, the occurrence of T will generate the relevant label as part of the observation, otherwise no label will be generated. Intuitively, the larger the set of visible transitions, the more observable the network reaction. However, the same label may be associated with several transitions. When all transitions in $\aleph$ are silent, we say that $\mathcal{V}$ is *blind* and $\wp(\aleph)$ is a *blind diagnostic problem*. As such, a blind viewer is incapable of observing any transition at all.

Example 4. Considering network *Net* depicted in Figure 3, along with the relevant automata outlined in Figure 2, a possible viewer $\mathcal{V}_N$ for *Net* is defined in Table I, where we assume label ε for all omitted transitions.

3.2. Observation

Informally, an observation $\mathcal{O}$ of $\aleph$ is the set of observable labels relevant to a reaction of $\aleph$, based on $\mathcal{V}$, typically under uncertainty conditions, as detailed in [28]. More precisely, $\mathcal{O}$ is a directed acyclic graph (DAG)

$$\mathcal{O} = (\Omega, \mathbb{E}) \qquad (1)$$

Table I. Viewer $\mathcal{V}_N$ for network *Net* (silent transitions are omitted).

Transition	Visible label
$T_1(b_1)$	o_1
$T_1(b_2)$	o_2
$T_2(p)$	l
$T_3(p)$	r

where $\Omega = \{\omega_1, \ldots, \omega_n\}$ is the set of nodes, each node being marked by a non-empty subset of the labels of $\mathcal{V}$, and $\mathbb{E}$ is the set of edges. A *precedence relationship* is defined between nodes, specifically, $\omega \prec \omega'$ means that $\mathcal{O}$ includes a path from ω to ω', while $\omega \preceq \omega'$ means either $\omega \prec \omega'$ or $\omega = \omega'$.

The observation $\mathcal{O}$ is said to be uncertain because of the multiplicity of the labels marking each node (*logical uncertainty*) and the partial ordering among nodes (*temporal uncertainty*).

Note the difference between what is observable, as stated by the viewer, and what is observed, as represented by the observation. The viewer establishes what (possibly empty) label is generated by each transition of $\aleph$. Of course, the degree of observability of $\aleph$ depends on the number of visible transitions (those associated with a non-empty label) and the degree of repetition of the same label for different transitions (for example, associating the same label *open* to several transitions).

Recall that, when reacting, $\aleph$ performs a sequence of transitions, namely a history h. The *signature* of h is the sequence of labels associated with the transitions of h based on a viewer $\mathcal{V}$, precisely

$$\langle \ell \mid T \in h, (T, \ell) \in \mathcal{V} \rangle \tag{2}$$

Ideally, the signature of h is expected to be perceived as the sequence of visible labels within it, called the *symptom* of h. In more real terms, owing to the possible multiplicity of the communication channels and to possible noise, what is actually perceived is not the symptom of h, but, rather, a relaxation of it, in fact, the observation $\mathcal{O}$. The reason for such a relaxation is twofold:

(1) each label ℓ associated with each transition of the history (ε included) is perceived as a set of candidate labels $\{\ell_1, \ldots, \ell_k\}$ including ℓ[‡];
(2) the absolute temporal order among the labels in the symptom is perceived as a partial order among their surrogates, the sets of candidate labels.

As such, an observation graph still includes the symptom of h. However, it also contains several other spurious symptoms, which may or may not be consistent with the behavior space of $\aleph$.

Example 5. Depicted in Figure 5 is the graph relevant to an observation $\mathcal{O}_N$ for network *Net*, based on viewer $\mathcal{V}_N$ defined in Table I. It is composed of four nodes, namely $\Omega = \{\omega_1, \ldots, \omega_4\}$.

[‡]When the set of candidate labels is the singleton $\{\varepsilon\}$, no node is included in the observation graph, since the label ε remains invisible.

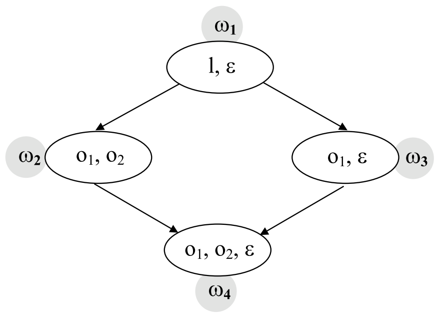

Figure 5. Observation $\mathcal{O}_N$ of *Net*.

Node ω_1 is marked by the set $\{l, \varepsilon\}$, meaning that either l or nothing was first generated by the reaction. The next actually generated observable label is one of the labels within the successive nodes, that is, ω_2 and ω_3, marked by the sets of labels $\{o_1, o_2\}$ and $\{o_1, \varepsilon\}$, respectively. Both of these nodes are to be considered before the last observable label, since they precede the last node ω_4, marked by $\{o_1, o_2, \varepsilon\}$.

3.3. Indexing observations

Since it is neither trivial nor efficient to reason about the observation graph as is, an additional graph is generated, called the *index space* of the temporal observation $\mathcal{O}$, namely *IndSpace*($\mathcal{O}$). The peculiarity of an index space is that each path, from the root to a final node, represents a mode in which labels may be chosen in the observation graph without violating the constraints imposed by the graph [28]. As such, once removed the ε labels from each path, the extension of *IndSpace*($\mathcal{O}$) is the complete set of symptoms implicitly specified by the observation graph. The computation of the index space of an observation $\mathcal{O} = (\Omega, \mathbb{E})$ is based on the following notions.

A *prefix* $\mathcal{P}$ of $\mathcal{O}$ is a subset of Ω where for all $\omega \in \mathcal{P}$ (there exists no $\omega' \in \mathcal{P}(\omega' \prec \omega)$). The set of *consumed nodes* up to $\mathcal{P}$ is

$$Cons(\mathcal{P}) = \{\omega \mid \omega \in \Omega, \omega' \in \mathcal{P}, \omega \preceq \omega'\} \tag{3}$$

The set of consumable nodes, called the *frontier* of $\mathcal{P}$, is

$$Front(\mathcal{P}) = \{\omega \mid \omega \in (\Omega - Cons(\mathcal{P})), \forall(\omega' \mapsto \omega) \in \mathbb{E}\ (\omega' \in Cons(\mathcal{P}))\} \tag{4}$$

The *prefix space* of $\mathcal{O}$ is the non-deterministic automaton

$$PrefSpace(\mathcal{O}) = (\mathbb{S}^{\mathrm{n}}, \mathbb{L}^{\mathrm{n}}, \mathbb{T}^{\mathrm{n}}, S_0^{\mathrm{n}}, S_{\mathrm{f}}^{\mathrm{n}}) \tag{5}$$

where $\mathbb{S}^{\mathrm{n}} = \{\mathcal{P} \mid \mathcal{P} \text{ is a prefix of } \mathcal{O}\}$ is the set of states, $\mathbb{L}^{\mathrm{n}}$ is the set of labels marking the nodes of $\mathcal{O}$, $S_0^{\mathrm{n}} = \emptyset$ is the initial state, $S_{\mathrm{f}}^{\mathrm{n}} \in \mathbb{S}^{\mathrm{n}}$, where $Cons(S_{\mathrm{f}}^{\mathrm{n}}) = \Omega$, is the final state, and $\mathbb{T}^{\mathrm{n}} : \mathbb{S}^{\mathrm{n}} \times \mathbb{L}^{\mathrm{n}} \mapsto 2^{\mathbb{S}^{\mathrm{n}}}$ is the transition function such that $\mathcal{P} \xrightarrow{\ell} \mathcal{P}' \in \mathbb{T}^{\mathrm{n}}$ if and only if, defining the operation

$$\mathcal{P} \oplus \omega = (\mathcal{P} \cup \{\omega\}) - \{\omega' \mid \omega' \in \mathcal{P}, \omega' \prec \omega\} \tag{6}$$

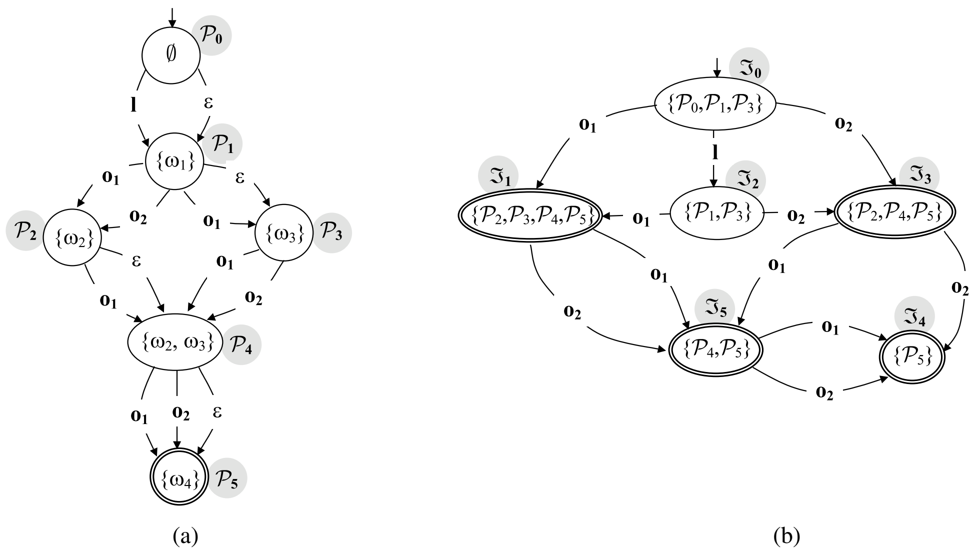

Figure 6. Prefix space (a) and index space (b) of observation $\mathcal{O}_N$ (displayed in Figure 5).

we have $\omega \in Front(\mathcal{P})$, $\ell \in \omega$ and $\mathcal{P}' = \mathcal{P} \oplus \omega$. Finally, *IndSpace*$(\mathcal{O})$ is the deterministic automaton equivalent to *PrefSpace*$(\mathcal{O})$.

Example 6. The prefix space of observation $\mathcal{O}_N$ (displayed in Figure 5) is shown in Figure 6(a). Each node is identified by a (possibly empty) subset of the nodes of $\mathcal{O}_N$, specifically, by a prefix of $\mathcal{O}_N$. Nodes are marked by $\mathcal{P}_i$, $i \in [0 .. 5]$, where $\mathcal{P}_0 = \emptyset$ is the root, while $\mathcal{P}_5 = \{\omega_4\}$ is the final node. Note how *PrefSpace*$(\mathcal{O}_N)$ is non-deterministic, owing both to ε-transitions and to multiple transitions leaving the same state and marked by the same label.

The index space of $\mathcal{O}_N$, obtained by transforming *PrefSpace*$(\mathcal{O}_N)$ into a deterministic equivalent automaton, is shown in Figure 6(b). According to the *subset construction* algorithm [29], each state of *IndSpace*$(\mathcal{O}_N)$ is identified by a proper subset of the states of the non-deterministic automaton. In particular, $\Im_0 = \{\mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_3\}$ is the root, while $\Im_1$, $\Im_3$, $\Im_4$ and $\Im_5$ are the final nodes.

Note how each path is a symptom consistent with $\mathcal{O}_N$. For instance, the temporal sequence $\langle o_1 \rangle$ corresponds to choosing ε in the first node ω_1 of the observation, o_1 in ω_2, ε in ω_3 and ε in ω_4[§]. In contrast, $\langle l \rangle$ is not a symptom, as $\Im_2$ is not final. This is consistent with the observation graph, as choosing l in ω_1 requires, for completing the observation, at least an additional choice between o_1 and o_2 in ω_2. A symptom of four observable labels corresponds to a choice where no ε is selected, for instance, $\langle l, o_1, o_2, o_2 \rangle$[¶].

[§]This choice generates the string $\langle \varepsilon, o_1, \varepsilon, \varepsilon \rangle$, which is equal to $\langle o_1 \rangle$, as the null label ε is irrelevant.
[¶]This is an example of spurious symptom that is inconsistent with the behavior space of *Net*.

Table II. Ruler $\mathcal{R}_N$ for network *Net* (normal transitions are omitted).

Transition	Fault label
$T_1(p)$	s
$T_2(b_1)$	f_1
$T_2(b_2)$	f_2

3.4. Ruler

A ruler establishes which transitions are to be considered faulty and the granularity of the diagnosis. $\mathcal{R}$ is defined by associating each transition T of each automaton in $\aleph$ with a fault label φ. If $\varphi \neq \varepsilon$, then the transition is *faulty* else T is *normal*.

Example 7. Considering network *Net* depicted in Figure 3, along with the relevant automata outlined in Figure 2, a possible ruler $\mathcal{R}_N$ for *Net* is defined in Table II, where we assume label ε for all omitted transitions. Specifically, fault s stands for *shorted*, while faults f_1 and f_2 denote *failed-to-open* for breakers b_1 and b_2, respectively.

3.5. Problem solution

What is the solution to a diagnostic problem? A diagnostic problem is symbolic in nature, and its solution is expected to involve a group of fault labels defined in the ruler. Let $\wp(\aleph) = (\aleph_0, \mathcal{V}, \mathcal{O}, \mathcal{R}, \mathcal{K})$ be the problem. We know that the network $\aleph$ is in state $\aleph_0$ when the reaction starts. We also know that the reaction of $\aleph$ results in the generation of observation $\mathcal{O}$, based on viewer $\mathcal{V}$. Thus, the history of $\aleph$ during the reaction is expected to be consistent with both $\mathcal{O}$ and $\mathcal{V}$. Generally speaking, several histories may be equally possible. According to the notion of a behavior space, each of such histories is a path in the behavior space $Bhv(\aleph, \aleph_0)$ that starts at $\aleph_0$ and ends at a final state. Hence, the whole set of histories consistent with $\mathcal{O}$ and $\mathcal{V}$ will be a subset of all the possible histories in the behavior space, namely $\|Bhv(\aleph, \aleph_0)\|$. As discussed above, each history implicitly entails a sequence of observable labels, precisely, the ordered set of labels (ε aside) associated by $\mathcal{V}$ with each transition of the history, namely the symptom of the history. This can be formalized by the product between a history and a viewer: the *product* of a history $h \in \|Bhv(\aleph, \aleph_0)\|$ and $\mathcal{V}$ is the sequence of observable labels

$$h \otimes \mathcal{V} = \langle \ell \mid T \in h, (T, \ell) \in \mathcal{V}, \ell \neq \varepsilon \rangle \tag{7}$$

Since each path in the index space $IndSpace(\mathcal{O})$, starting at the initial state and ending at a final state (a symptom), is a possible trace of the reaction of $\aleph$, we conclude that h is consistent with $\mathcal{O}$ and $\mathcal{V}$ when the product of h with $\mathcal{V}$ is a symptom, namely when $(h \otimes \mathcal{V}) \in \|IndSpace(\mathcal{O})\|$. If so, h is called a *candidate history*. Thus, the set of candidate histories relevant to $\wp(\aleph)$ is defined as

$$Hyst(\wp(\aleph)) = \{h \mid h \in \|Bhv(\aleph, \aleph_0)\|, (h \otimes \mathcal{V}) \in \|IndSpace(\mathcal{O})\|\} \tag{8}$$

The set of candidate histories may be unbounded when the behavior space contains an infinite number of histories due to internal cycles. On the other hand, the candidate histories are only an intermediate result of the diagnostic process. The actual diagnostic output, namely the *solution* of the diagnostic problem, is the set of *candidate diagnoses*, a surrogate of the candidate histories. Intuitively, each candidate history h entails a candidate diagnosis δ, this being the set of fault labels associated with the transitions of h in the ruler $\mathcal{R}$ (ε aside). This can be formalized by the product between a history and a ruler:

$$\delta = h \otimes \mathcal{R} = \{\varphi \mid T \in h,\ (T, \varphi) \in \mathcal{R},\ \varphi \neq \varepsilon\} \tag{9}$$

Due to the set-theoretic structure of δ, possible replications of the same fault label are implicitly removed from δ. Hence, a candidate diagnosis is bounded by the whole (finite) set of faulty labels in the ruler. Also, two different histories may entail the same diagnosis, as the latter only depends on the fault labels. Of course, the diagnosis entailed by a history involving only normal transitions is, in fact, an empty set. In other words, an empty diagnosis denotes a normal reaction of the network.

The diagnostic-problem *solution* is therefore the whole set Δ of candidate diagnoses entailed by the set of candidate histories, namely

$$\Delta(\wp(\aleph)) = \{\delta \mid \delta = h \otimes \mathcal{R},\ h \in \mathit{Hyst}(\wp(\aleph))\} \tag{10}$$

In contrast with the set of candidate histories, the solution Δ of the problem is always a finite set, because of the finiteness of the set of fault labels.

Example 8. With reference to network *Net* shown in Figure 3, consider the diagnostic problem

$$\wp(\mathit{Net}) = (\mathit{Net}_0, \mathcal{V}_N, \mathcal{O}_N, \mathcal{R}_N, \mathcal{K}_N)$$

where $\mathit{Bhv}(\mathit{Net}, \mathit{Net}_0)$ is displayed in Figure 4, $\mathcal{V}_N$ in Table I, $\mathcal{O}_N$ in Figure 5, $\mathit{IndSpace}(\mathcal{O}_N)$ in Figure 6 and $\mathcal{R}_N$ in Table II. Also, assume that $\mathcal{K}_N$ does not embody any compiled knowledge. According to Equation (8), the set of candidate histories for our problem is

$$\mathit{Hyst}(\wp(\mathit{Net})) = \{h \mid h \in \|\mathit{Bhv}(\mathit{Net}, \mathit{Net}_0)\|,\ (h \otimes \mathcal{V}_N) \in \|\mathit{IndSpace}(\mathcal{O}_N)\|\}.$$

Evidence from Figures 4 and 6 indicates that the subset of symptoms in $\|\mathit{IndSpace}(\mathcal{O}_N)\|$ that is consistent with the histories in the behavior space is $\{\langle o_1\rangle, \langle o_2\rangle, \langle l, o_2\rangle, \langle o_1, o_2\rangle, \langle o_2, o_1\rangle\}$. These symptoms are entailed by the candidate histories

$$\begin{aligned}\mathit{Hyst}(\wp(\mathit{Net})) = \{&\langle T_1(p), T_1(b_1), T_2(b_2), T_5(p)\rangle, \langle T_1(p), T_2(b_2), T_1(b_1), T_5(p)\rangle,\\ &\langle T_1(p), T_2(b_2), T_5(p), T_1(b_1)\rangle, \langle T_1(p), T_1(b_2), T_2(b_1), T_4(p)\rangle,\\ &\langle T_1(p), T_2(b_1), T_1(b_2), T_4(p)\rangle, \langle T_1(p), T_2(b_1), T_4(p), T_1(b_2)\rangle,\\ &\langle T_2(p), T_1(b_2)\rangle, \langle T_1(p), T_1(b_1), T_1(b_2)\rangle, \langle T_1(p), T_1(b_2), T_1(b_1)\rangle\}\end{aligned}$$

Based on Equation (10), the set of candidate diagnoses, namely the solution of $\wp(\mathit{Net})$, is

$$\Delta(\wp(\mathit{Net})) = \{\emptyset, \{s\}, \{s, f_1\}, \{s, f_2\}\}.$$

Note how different candidate histories generate the same candidate diagnosis. The candidate diagnoses can be paraphrased as follows: $\delta_1 = \emptyset$, *nothing misbehaved*; $\delta_2 = \{s\}$, *a short circuit occurred and the reaction was normal*; $\delta_3 = \{s, f_1\}$, *a short circuit occurred and breaker* b_1 *failed to open*; $\delta_4 = \{s, f_2\}$, *a short circuit occurred and breaker* b_2 *failed to open*. However, since all candidate diagnoses have the same ontological status, there is no formal means of choosing one diagnosis over another. Instead, a ranking of the candidate diagnoses can be possibly defined based on additional application-oriented criteria.

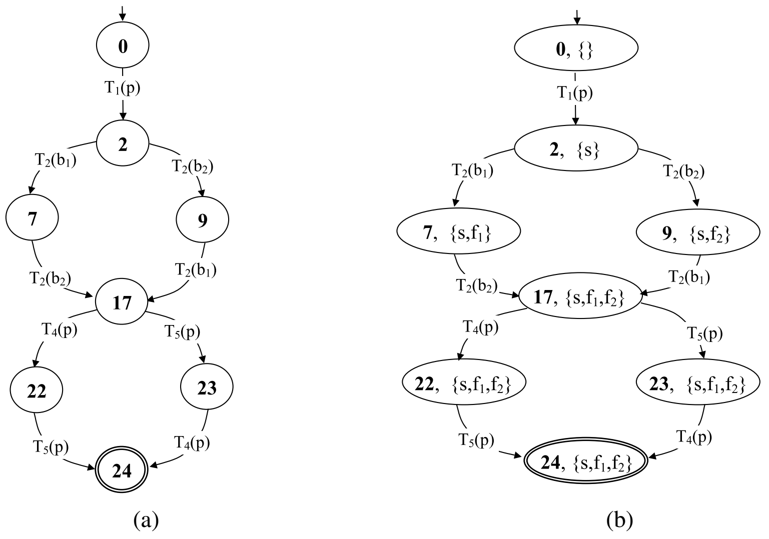

Figure 7. Behavior for *Net* (a) and relevant abduction (b).

3.6. Compiled knowledge

Compiled knowledge pertains to a certain initial state of $\aleph$. We assume that $\mathcal{K}$ is the knowledge relevant to the initial state $\aleph_0$ expressed in $\wp(\aleph)$. More precisely, $\mathcal{K}$ is the subset of the compiled-knowledge base that is relevant to $\aleph_0$, rather than the whole compiled knowledge relevant to $\aleph$. Three classes of compiled knowledge are envisaged, which are instantiated by relevant *knowledge graphs*, namely:

- *behavior*, a graph whose extension is a subset of the extension of the behavior space;
- *abduction*, a graph whose extension is a subset of the extension of the behavior space and where each node is decorated with the set of faults progressively encompassed by the relevant histories, in accordance with a ruler;
- *map*, a graph where each path is the sequence of observable labels relevant to histories of $\aleph$, in accordance with a viewer, and each node is marked by a set of diagnoses, in accordance with a ruler.

Example 9. With reference to the behavior space of *Net* shown in Figure 4, displayed in Figure 7(a) is a behavior of *Net* corresponding to the histories involving both transitions $T_2(b_1)$ and $T_2(b_2)$. Nodes are marked by the identifiers of nodes in Figure 4. Note how, unlike the behavior space, even if idle, the initial node 0 is not final because the empty history does not meet the requirement for this graph. The behavior in Figure 7 exhibits two peculiarities: (i) the lack of repeated nodes; and (ii) the lack of cycles. Both peculiarities are implied by the lack of cycles in the corresponding behavior space. Instead, when the latter is cyclic, the iteration of one cycle more than once generates several instances

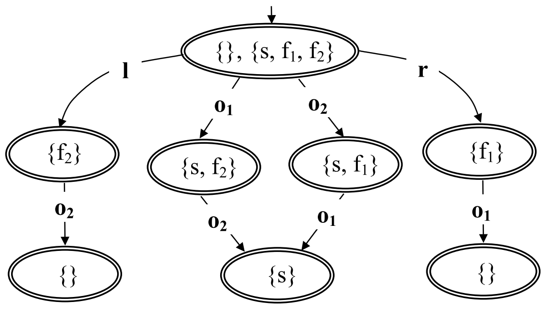

Figure 8. Map for *Net*.

of the same node, or even includes cycles in its turn. Therefore, the behavior space may also possibly involve an unbounded number of histories. As expected, each history in Figure 7 is also a history in Figure 4.

The notion of an abduction is similar to that of a behavior, inasmuch as in both graphs a path is a history of the system. However, unlike a behavior, an abduction carries explicit diagnostic information.

Example 10. The abduction of *Net* corresponding to the behavior in Figure 7(a) is depicted in Figure 7(b), where ruler $\mathcal{R}_N$ in Table II is assumed. Each node is marked by a pair (σ, δ), where σ is a node of the behavior space, while δ is the set of faults relevant to each path connecting the root of the graph to the node. For example, in node $(17, \{s, f_1, f_2\})$, δ includes s, f_1, and f_2, as both paths from the root involve faulty transitions $T_1(p)$, $T_2(b_1)$ and $T_2(b_2)$.

Although not applicable to the abduction in Figure 7, generally speaking, an abduction may include several nodes with the same σ and different δ, as nodes are identified by both fields. Consequently, given a behavior $\mathcal{B}$ of $\aleph$, the corresponding abduction $\mathcal{A}$ is not merely a decoration of $\mathcal{B}$, as nodes of $\mathcal{B}$ may be replicated within $\mathcal{A}$ with different δ. For example, considering the behavior, if the two paths from the root to node 17 generate two different diagnoses, namely δ_1 and δ_2, then the abduction in Figure 7(b) will involve two different nodes, namely $(17, \delta_1)$ and $(17, \delta_2)$.

Since, by definition, a diagnosis is the set of faults relevant to a history of *Net*, the field δ associated with each final node is a possible (candidate) diagnosis of *Net*. In our example, only the node marked by 24 is final, thereby the set of candidate diagnoses is the singleton $\Delta = \{\delta\}$, where $\delta = \{s, f_1, f_2\}$, which is consistent with our assumption on the histories of the abduction.

The notion of a map is similar to that of an abduction, as each node carries diagnostic information, specifically, a set of sets of faults. However, in contrast with both behaviors and abductions, each edge of a map is marked by an observable label rather than a component transition. This means that both a viewer and a ruler are to be assumed.

Example 11. The map of *Net* corresponding to the behavior space of *Net* shown in Figure 4 is displayed in Figure 8. Each node of the map is qualified by a set of diagnoses, based on ruler $\mathcal{R}_N$ defined in Table II. Each edge is marked by an observable label, based on the viewer $\mathcal{V}_N$ in Table I.

A path from the root to a final node[‖] corresponds to the sequence of observable labels entailed by a set of histories of *Net*. For example, path $\langle o_1, o_2\rangle$ is entailed by the single history $\langle T_1(p), T_1(b_1), T_1(b_2)\rangle$ (see Figure 4), involving just fault s. Instead, the one-step path $\langle o_1\rangle$ is entailed by three different histories, namely $\langle T_1(p), T_1(b_1), T_2(b_2), T_5(p)\rangle$, $\langle T_1(p), T_2(b_2), T_5(p), T_1(b_1)\rangle$ and $\langle T_1(p), T_2(b_2), T_1(b_1), T_5(p)\rangle$, involving the set of faults $\{s, f_2\}$. In either case, the set of faults entailed by the corresponding histories is equal to the diagnosis marking the final node in the map.

A more general case is represented by the root, which involves two diagnoses: the empty diagnosis and $\{s, f_1, f_2\}$. The former does not involve any reaction at all, while the latter corresponds to a short circuit on the protected device with a misbehaving reaction involving the failure of both breakers. Such a reaction, however, does not generate any observable label.

3.7. Knowledge reuse

Reuse of knowledge graphs is a major goal of the diagnostic environment. Specifically, reusing a diagnostic problem $\wp$ for solving a new problem $\wp'$ means exploiting the knowledge graphs generated for solving $\wp$. The fundamental question is: *Under which circumstances can the knowledge graphs generated for solving* $\wp$ *be reused for solving* $\wp'$? To answer this question, we introduce a relationship called *subsumption*, that is applicable to observations, viewers, rulers and diagnostic problems. Formal results indicate that problem subsumption is the condition for knowledge reuse [27].

3.7.1. Observation subsumption

Let $\mathcal{O}$ and $\mathcal{O}'$ be two observations. We say that $\mathcal{O}$ *subsumes* $\mathcal{O}'$, written $\mathcal{O} \Supset \mathcal{O}'$, when the set of symptoms of $\mathcal{O}$ contains the set of symptoms of $\mathcal{O}'$, in other terms,

$$\mathcal{O} \Supset \mathcal{O}' \iff \|IndSpace(\mathcal{O})\| \supseteq \|IndSpace(\mathcal{O}')\| \tag{11}$$

Intuitively, assuming the same viewer for both observations, if $\mathcal{O} \Supset \mathcal{O}'$, then $\mathcal{O}$ is consistent with all of the symptoms of $\mathcal{O}'$. Consequently, the constraints imposed by $\mathcal{O}$ are less stringent than those imposed by $\mathcal{O}'$, as each possible symptom of $\mathcal{O}$ represents a chance of consistency with an additional set of histories for the network and, possibly, an additional set of candidate diagnoses.

Example 12. Shown in Figure 9(a) is an observation $\mathcal{O}'_N$ for *Net*, which is subsumed by the observation $\mathcal{O}_N$ in Figure 5, namely $\mathcal{O}_N \Supset \mathcal{O}'_N$. Comparing the respective index spaces (see Figure 6), it is easy to check the containment of the symptoms, namely $\|IndSpace(\mathcal{O}_N)\| \supset \|IndSpace(\mathcal{O}'_N)\|$.

3.7.2. Viewer and ruler subsumption

The notion of subsumption can be extended to viewers and rulers alike. Since both viewers and rulers are essentially defined in the same way, namely as a mapping from the domain of network transitions and a domain of labels (including ε), it suffices to introduce viewer subsumption.

[‖]Incidentally, in this example, all nodes are final.

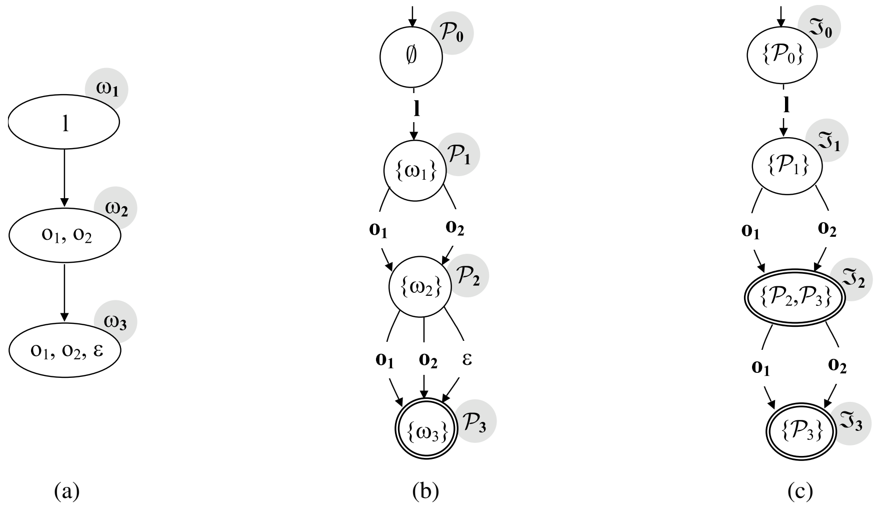

Figure 9. Observation $\mathcal{O}'_N$ for *Net* (a), *PrefSpace*$(\mathcal{O}'_N)$ (b) and *IndSpace*$(\mathcal{O}'_N)$ (c).

Let $\mathcal{V}$ and $\mathcal{V}'$ be two viewers relevant to the isomorphic networks $\aleph$ and $\aleph'$ (in the simplest case, $\aleph = \aleph'$). Let $\boldsymbol{\Lambda}$ and $\boldsymbol{\Lambda}'$ be the domains of labels (including ε) relevant to $\mathcal{V}$ and $\mathcal{V}'$, respectively. We say that $\mathcal{V}$ *subsumes* $\mathcal{V}'$, denoted $\mathcal{V} \Supset \mathcal{V}'$, if and only if the following two conditions hold:

(1) the set of visible transitions involved in $\mathcal{V}'$ is isomorphic to a subset of the visible transitions involved in $\mathcal{V}$;
(2) for each label ℓ associated in $\mathcal{V}$ with a visible transition that has a corresponding visible transition in $\mathcal{V}'$ associated with ℓ', the set of visible transitions associated with ℓ in $\mathcal{V}$ is isomorphic to a subset of the visible transitions associated with ℓ' in $\mathcal{V}'$.

In order to pursue reuse, we need to define a renaming function between labels of viewers. Let ℓ be a label in $\boldsymbol{\Lambda}$. The *renaming* of ℓ within the context of $\mathcal{V}$ and $\mathcal{V}'$ is a label ℓ' in $\boldsymbol{\Lambda}'$, defined as follows:

$$Ren(\ell, \mathcal{V}, \mathcal{V}') = \ell', \quad \text{where } (T, \ell) \in \mathcal{V},\ (T', \ell') \in \mathcal{V}',\ T \between T' \tag{12}$$

The subsumption relationship (and the renaming function) can be straightforwardly extended to rulers by substituting $\mathcal{V}$ and $\mathcal{V}'$ with $\mathcal{R}$ and $\mathcal{R}'$, respectively.

Example 13. Consider viewer $\mathcal{V}_N$ in Table I for *Net* and the new viewer $\mathcal{V}'_N$ defined in Table III. Note how $\mathcal{V}'_N$ differs from $\mathcal{V}_N$. First, $T_3(p)$ is no longer visible in $\mathcal{V}'_N$. Second, $T_1(b_1)$ and $T_2(b_2)$ are associated with the same label *open* (thereby making *Net* less observable). Finally, the label associated with $T_2(p)$ has changed to *left*. Clearly, $\mathcal{V}_N \Supset \mathcal{V}'_N$. In fact, according to our definition: (1) the set of visible transitions involved in $\mathcal{V}'_N$ is isomorphic to (is, in fact) a subset of the visible transitions involved in $\mathcal{V}_N$; and (2) for each label ℓ associated in $\mathcal{V}_N$ with a visible transition that has a corresponding visible transition in $\mathcal{V}'_N$ associated with ℓ' (for instance, transition $T_1(p)$ associated with o_1 in $\mathcal{V}_N$

Table III. Viewer $\mathcal{V}'_N$ for network *Net* (silent transitions are omitted).

Transition	Visible label
$T_1(b_1)$	*open*
$T_1(b_2)$	*open*
$T_2(p)$	*left*

has the corresponding transition in $\mathcal{V}'_N$ associated with *open*), the set of visible transitions associated with ℓ in $\mathcal{V}_N$ is isomorphic to (is, in fact) a subset of the visible transitions associated with ℓ' in $\mathcal{V}'_N$ (actually, $\{T_1(b_1)\}$ is a subset of $\{T_1(b_1), T_1(b_2)\}$). Moreover, based on Equation (12), we have $Ren(o_1, \mathcal{V}_N, \mathcal{V}'_N) = open$, $Ren(l, \mathcal{V}_N, \mathcal{V}'_N) = left$ and $Ren(r, \mathcal{V}_N, \mathcal{V}'_N) = \varepsilon$ (recall that silent transitions are omitted in the tables).

3.7.3. *Observation projection*

Let $\mathcal{O}$ be an observation for a network $\aleph$, and $\mathcal{V}$ a relevant viewer. Let $\mathcal{V}'$ be a different viewer for $\aleph$, such that $\mathcal{V} \Supset \mathcal{V}'$. The projection of a node ω in $\mathcal{O}$ on $\mathcal{V}'$, namely $\omega_{[\mathcal{V}']}$, is a node ω' that includes a label ℓ' for each label ℓ in ω, where

$$\ell' = \begin{cases} \varepsilon & \text{if } \ell = \varepsilon \\ Ren(\ell, \mathcal{V}, \mathcal{V}') & \text{otherwise} \end{cases} \tag{13}$$

With ω' being a set, duplicated labels $\ell' \in \omega'$ are removed. The projection of $\mathcal{O}$ on $\mathcal{V}'$, denoted $\mathcal{O}_{[\mathcal{V}']}$, is an observation $\mathcal{O}'$ where, for each node ω in $\mathcal{O}$, $\mathcal{O}'$ includes a node $\omega' = \omega_{[\mathcal{V}']}$ (only when $\omega' \neq \{\varepsilon\}$, otherwise ω' is omitted).

Example 14. Consider viewers $\mathcal{V}_N$ and $\mathcal{V}'_N$ defined in Tables I and III, respectively. Displayed in Figure 10 are the observation $\mathcal{O}_N$ (Figure 10(a)) relevant to viewer $\mathcal{V}_N$ and the projection $\mathcal{O}_{N[\mathcal{V}'_N]}$ (Figure 10(b)). Accordingly, labels in $\mathcal{V}_N$ have been renamed into corresponding labels in $\mathcal{V}'_N$, with duplicate removal. For instance, the set of labels $\{o_1, o_2\}$ has been transformed into the singleton $\{open\}$, as $Ren(o_1, \mathcal{V}_N, \mathcal{V}'_N) = Ren(o_2, \mathcal{V}_N, \mathcal{V}'_N) = open$.

3.7.4. *Problem subsumption*

We can now answer the question raised in Section 3.7: the solution of a diagnostic problem $\wp_2$ may be supported by the knowledge relevant to the solution of a previous diagnostic problem $\wp_1$, provided that $\wp_1$ subsume $\wp_2$. Intuitively, if such a subsumption holds, the knowledge necessary for solving $\wp_2$ is somewhat incorporated within the knowledge generated for solving $\wp_1$.

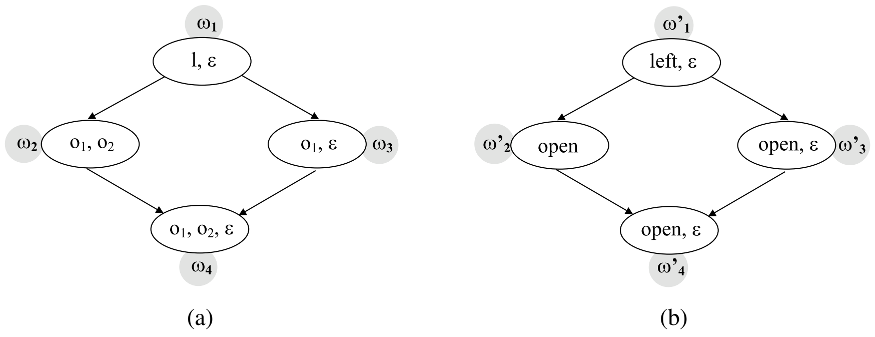

Figure 10. Observation $\mathcal{O}_N$ (a) and relevant projection $\mathcal{O}_{N[\mathcal{V}'_N]}$ (b).

Let $\wp_1 = (\Psi_{01}, \mathcal{V}_1, \mathcal{O}_1, \mathcal{R}_1, \mathcal{K}_1)$ and $\wp_2 = (\Psi_{02}, \mathcal{V}_2, \mathcal{O}_2, \mathcal{R}_2, \mathcal{K}_2)$ be two problems for two isomorphic networks (possibly the same network) $\aleph_1$ and $\aleph_2$, respectively. We say that $\wp_1$ *weakly subsumes* $\wp_2$, denoted $\wp_1 \sqsupseteq \wp_2$, if and only if $\mathcal{V}_1$ is subsumed by $\mathcal{V}_2$ and $\mathcal{O}_1$ subsumes the projection of $\mathcal{O}_2$ on $\mathcal{V}_1$:

$$\wp_1 \sqsupseteq \wp_2 \iff (\mathcal{V}_1 \Subset \mathcal{V}_2,\, \mathcal{O}_1 \Supset \mathcal{O}_{2[\mathcal{V}_1]}) \tag{14}$$

We say that $\wp_1$ (*strongly*) *subsumes* $\wp_2$, denoted $\wp_1 \Supset \wp_2$, if and only if $\wp_1 \sqsupseteq \wp_2$ and $\mathcal{R}_1 \Supset \mathcal{R}_2$, that is,

$$\wp_1 \Supset \wp_2 \iff (\mathcal{V}_1 \Subset \mathcal{V}_2,\, \mathcal{O}_1 \Supset \mathcal{O}_{2[\mathcal{V}_1]},\, \mathcal{R}_1 \Supset \mathcal{R}_2) \tag{15}$$

The definition of problem subsumption may sound odd, especially because of the inverted subsumption relationship between viewers. Intuitively, assuming for simplicity $\aleph_1 = \aleph_2$, it allows the observation $\mathcal{O}_1$ to be *no* more constrained than $\mathcal{O}_2$. Consequently, the behavior relevant to $\wp_1$ will incorporate all of the histories of the behavior relevant to $\wp_2$. Thus, the behavior of $\wp_1$ may be reused to solve $\wp_2$. In addition, if strong subsumption holds, knowledge reusability may be extended to the graphs involving diagnostic information (abductions and maps).

Example 15. With reference to network *Net* displayed in Figure 3, consider two different diagnostic problems, $\wp_1(Net) = (Net_0, \mathcal{V}'_N, \mathcal{O}^1_N, \mathcal{R}_N, K^1_N)$ and $\wp_2(Net) = (Net_0, \mathcal{V}_N, \mathcal{O}^2_N, \mathcal{R}_N, K^2_N)$, where $\mathcal{V}_N$ and $\mathcal{V}'_N$ are defined in Tables I and III, respectively, $\mathcal{O}^1_N$ and $\mathcal{O}^2_N$ are displayed in Figure 11(a) and $\mathcal{R}_N$ is defined in Table II. Example 13 shows that $\mathcal{V}'_N \Subset \mathcal{V}_N$. Based on Figure 11, we can check that $\mathcal{O}^1_N \Supset O^2_{N[\mathcal{V}'_N]}$. In fact, the index space of $\mathcal{O}^1_N$ (Figure 11(c), left) contains both symptoms relevant to the index space of $O^2_{N[\mathcal{V}'_N]}$ (Figure 11(c), right), namely $\langle open, open\rangle$ and $\langle open, open, open\rangle$. Therefore, $\mathcal{O}^1_N \Supset O^2_{N[\mathcal{V}'_N]}$. Since the ruler $\mathcal{R}_N$ is the same, according to Equation (15), we conclude that $\wp_1(Net)$ strongly subsumes $\wp_2(Net)$, namely $\wp_1(Net) \Supset \wp_2(Net)$. As such, in order to solve $\wp_2(Net)$, we can reuse any of the knowledge graphs generated for solving $\wp_1(Net)$.

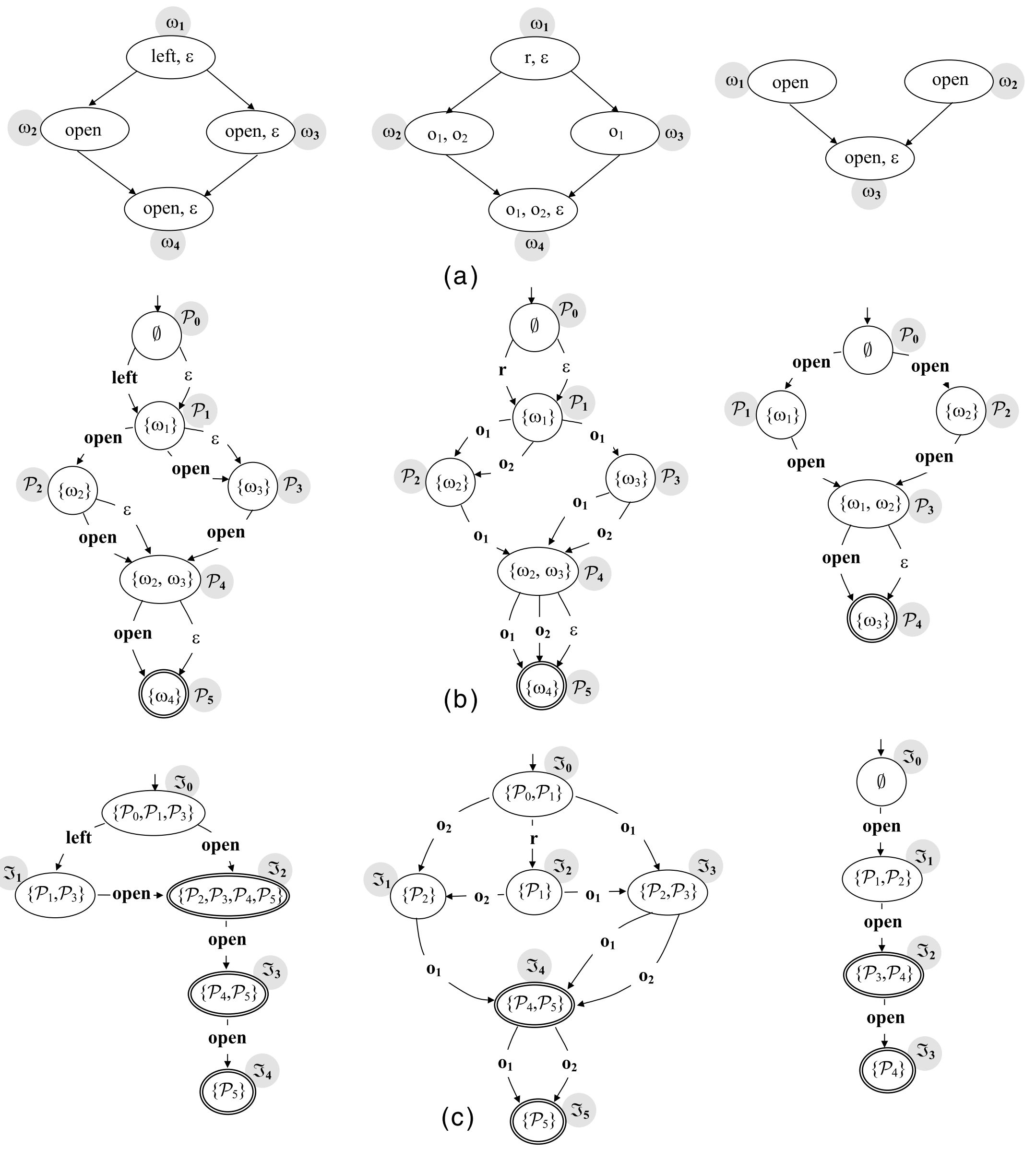

Figure 11. Observations $\mathcal{O}_N^1$, $\mathcal{O}_N^2$, $\mathcal{O}_{N[\mathcal{V}_N']}^2$ (a), and relevant prefix spaces (b) and index spaces (c).

4. REQUIREMENTS AND SOFTWARE ARCHITECTURE

In the previous sections we have introduced the task of diagnosis of CANs from a conceptual point of view. In this section we outline the requirements and the architecture of the software system that the diagnostic environment is based on. The design of the diagnostic environment was conceived on five conceptual requirements, namely the following.

(1) Both observability and abnormality properties of a system shall be dynamically bound to the actual diagnostic problem, rather than statically to the model of the involved system.
(2) The diagnostic problem shall involve uncertain observations.
(3) The compositional model of the system shall be possibly compiled off-line to generate diagnostic-oriented knowledge aimed at speeding up on-line diagnosis.
(4) If model compilation has generated knowledge off-line, then on-line diagnosis shall be confined to a pattern-matching activity.
(5) The knowledge generated on-line for solving a diagnostic problem shall be possibly reused when solving different diagnostic problems.

These requirements were translated into a number of constraints on the architecture of the software system, specifically the following.

- The diagnostic environment shall include an interface supporting the formal specification of CANs and relevant diagnostic problems.
- The diagnostic environment shall include a knowledge base.
- The diagnostic environment shall provide a translator of both system and problem specifications (stated in formal language) into structures of the knowledge base.
- The diagnostic environment shall include a preprocessor that generates compiled knowledge off-line with the support of the knowledge base.
- The diagnostic environment shall include a diagnostic-problem solver supported by the knowledge base.
- The communication between processes shall be based on a standard protocol and the module relevant to the knowledge base shall be available as a network service.

The last constraint is aimed at a flexible architecture and at a parallel computation on different machines.

Outlined in Figure 12 is the architecture of the diagnostic environment, which consists of four main components, namely:

- the compiler of the specification language into knowledge base data structures;
- the manager of the knowledge base;
- the preprocessor, for the off-line generation of diagnostic knowledge;
- the actual diagnostic engine, for solving diagnostic problems.

Conceptually, the diagnostic environment is stratified on two layers:

- the knowledge management for persistent storage;
- the knowledge management for problem solving.

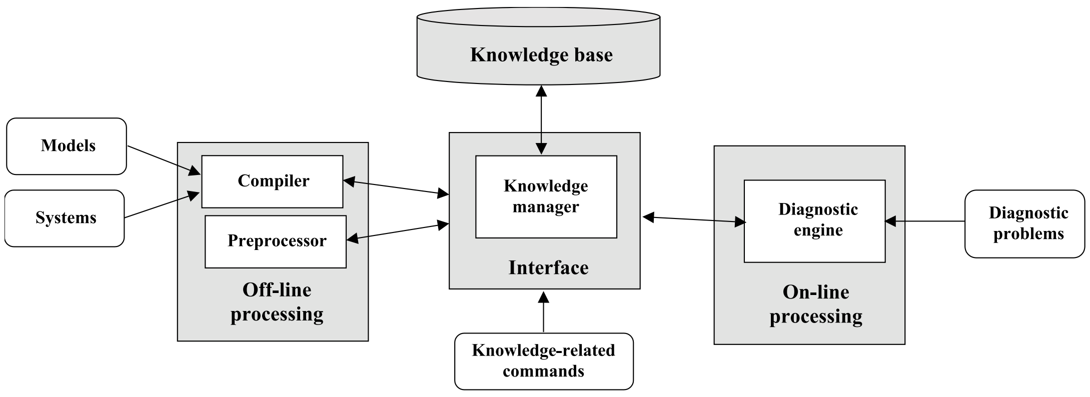

Figure 12. Architecture of the diagnostic environment.

As to the implementation, performance and portability issues led to the choice of the standard C (ISO C90) programming language. The chosen operating system was GNU/Linux. The latter provides the tools for compiler construction, including *Flex* (scanner generator) and *Bison* (parser generator), as well as the *PostgreSQL* database management system (DBMS), which supported the development of the knowledge base.

5. KNOWLEDGE-BASE DESIGN

The ability of reusing knowledge implies the ability of retrieving chunks of knowledge that are suitable for solving new problems. This translates, on the one hand, into the need for a persistent storage of knowledge (the so-called knowledge base mentioned in the previous section), and, on the other hand, into the need for proper knowledge-indexing mechanisms. Several design alternatives were explored to fulfill these needs. First of all, a relational database was designed and implemented that included both front-end data (inherent to models, systems and problems), and the whole compiled knowledge. Specifically, each node and edge of every graph (behavior, abduction, map, observation and index space) was saved in the database. However, the experiments exercising this implementation highlighted that data-transfer was exceedingly time-consuming (up to half an hour to save a graph on a PC supporting some thousands transactions per second). In fact, since a transaction was needed for saving each edge/node of the graph, millions of transactions were performed for large graphs.

Then the opposite design was tried, that is, all of the information was saved in the File System under the form of text files. This improved the performance of data transfer, since each graph was saved and retrieved as a whole, but degraded the performances of reusable-knowledge indexing. In fact, since the powerful DBMS mechanisms were not available, proper platform-dependent functions were coded, whose code was error-prone and difficult to maintain, and whose execution was slow.

The design that was finally chosen was a hybrid one, taking advantage of both the DBMS indexing mechanisms, so as to obtain good time performances in identifying reusable chunks of knowledge, and of text files for the actual storage of reusable knowledge, so as to obtain good time performances in data transfer. According to this choice, knowledge generated either off-line or on-line and also index spaces are saved under the form of text files. A specific data structure has been proposed for each kind of graph (behavior, abduction, map and index space), which mirrors the corresponding internal data structure used by the diagnostic engine. Therefore, once either the preprocessor or the diagnostic engine has produced a graph, it simply ‘downloads’ the relevant data structure into a file, and, dually, when the diagnostic engine is suggested by the knowledge manager module to exploit an existing graph, it ‘uploads’ the relevant data structure from a file.

The knowledge-indexing information is contained in a database, which is both maintained and exploited by the knowledge manager module. The main task of the knowledge manager is to find whether there exists any chunk of knowledge that could be reused for solving a new given problem. The database has been designed to support the operation of the knowledge manager, i.e. first of all to enable this task. In particular, the relational database tables are meant to represent all compiled domain models as well as all solved problems, including the compiled versions of the descriptions of the system, viewer, ruler and observation. Moreover, the same tables create a correspondence between the compiled description of the problem and the data structures produced for solving the problem, i.e. the knowledge graphs whose textual descriptions are contained in the File System. This correspondence implements the mechanism for knowledge-indexing.

The database is structured onto two hierarchical layers. The top layer, which has a public schema, contains the compiled descriptions of all of the systems and models compiled so far (plus the references to the files in the File System that persistently contain the native descriptions of such systems and models given by the user as input to the off-line processing module). Each time the compiled description of a new system has been generated, in addition to saving it in the top layer of the database, a new File System directory for this individual system is created wherein the native description is saved.

The bottom layer of the database is specific to each compiled system. Therefore, there is a bottom-layer database for each system, which contains the information inherent to the compiled knowledge of such a system. Each time a new compiled graph (behavior, abduction or map) inherent to a system is created, the relevant (indexing) information is saved in the database specific to the system while the graph is saved in a file, within the directory specific to the system.

The two-tier structure of the database was adopted for efficiency reasons, so as, in case the size of the compiled knowledge of a system is increased, the time performances of the queries inherent to the other systems are not affected.

Some relations of each system-specific database are listed in the following:

```
Problem (id, initial_state, id_viewer, id_ruler, id_observation, id_solution)

Observation (id, id_index_space)

Index_Space(id, length, num_nodes, filename)

Behavior (id, id_problem, initial_state, id_viewer, filename)

Abduction (id, id_problem, id_behavior, id_ruler, id_viewer, filename)

Map (id, id_problem, id_abduction, id_ruler, id_viewer, filename)
```

The attribute names are self-explaining. The *filename* attribute provides the name of the file in the File System that contains the relevant graph (index space, behavior, abduction or map). Each of these graphs is either the solution to a problem, univocally identified by *id_problem*, or a space graph, in which case the *id_problem* is assigned a special value. Each tuple of the *Problem* relation describes a problem that has already been solved, whose observation is identified by *id_observation*. Relation *Observation* provides the correspondence between the value of *id_observation* and the identifier of the index space of the observation itself. Finally, the relevant tuple of the *Index_Space* relation points to the file wherein the index space graph is stored. The same tuple provides the number of nodes of the index space and its 'length' (that is, the length of its longest path), both pieces of information that are exploited by the algorithm for the retrieval of reusable knowledge.

The attribute *id_solution* of *Problem* univocally identifies the set of diagnoses that constitutes the solution of the relevant problem. This solution is described by further relations that create the correspondences between each solution and several diagnoses, and between each diagnoses and several fault labels. In the case where a new problem equals an old one, this solution is given without extracting it from any graph.

Out of run-time efficiency, each system-specific database contains a view encompassing all of the compiled graphs, where each graph is endowed with the value of a new attribute (*type*) that univocally identifies the type of the graph itself (map space, map, abduction space, abduction, behavior space and behavior). Each tuple of the view describes a compiled graph and includes (all and only) the pieces of information that are exploited by Algorithm 3 (see Section 7.2) for retrieving reusable knowledge. This algorithm, before repeatedly performing subsumption checks, invokes an SQL query to order the content of the view based on three attribute values (the *type* of the graph, the *length* and *num_nodes* of the index-space of the observation relevant to the problem solved by the graph).

6. *DELTA* SPECIFICATION LANGUAGE

An essential conceptual tool of the diagnostic environment is the specification language, named *DELTA*, which allows users to define and catalog a variety of models, systems and problems**. *DELTA* was designed to substantiate in concrete statements the formal entities of the diagnostic approach. Moreover, it supports hierarchical compositional modeling. Three *compilation units* are allowed, namely the *domain*, *system* and *problem* units.

To introduce the language, consider the reference application of power transmission lines defined in Section 2. In Figure 13 the *DELTA* specification of the communicating automata displayed in Figure 2 is outlined, namely *Breaker* and *Protection*, along with a relevant clusterization. Specifically, a *domain* called *PowerTransmission* is declared. The *Breaker* automaton is defined by the first *component model* clause, which embodies the set of events, the set of input/output terminals, the set of states and the set of transitions. Also, a default initial state, whose role will be clarified below, is indicated by the *start* keyword. Note how each transition is defined: first the name of the transition, then the triggering event at the relevant input terminal (between the *when* and *do* keywords), followed by the actual state change

**We do not consider *DELTA* as the ideal interface of the diagnostic environment but, rather, an intermediate notation generated by a graphic-based human–computer interface, called Δ, which is, however, beyond the scope of this paper.

```
domain PowerTransmission;

component model Breaker {
    event z, f;
    input I;
    output O;
    state 0, 1;
    transition {
        T1: when (z @ I) do 0 -> 1 ();
        T2: when (z @ I) do 0 -> 0 (f @ O);
    }
    start 0;
}

component model Protection {
    event s, z, f;
    input I1, I2, I3, I4;
    output O1, O2, O3, O4;
    state 0, 1, 2, 3;
    transition {
        T1: when (s @ in) do 0 -> 1 (z @ O1), (z @ O2);
        T2: when (z @ I3) do 0 -> 2 (z @ O2);
        T3: when (z @ I4) do 0 -> 3 (z @ O1);
        T4: when (f @ I1) do 1 -> 1 (z @ O3);
        T5: when (f @ I2) do 1 -> 1 (z @ O4);
        T6: when (f @ I2) do 2 -> 2 (z @ O4);
        T7: when (f @ I1) do 3 -> 3 (z @ O3);
    }
    start 0;
}

cluster model ProtectedLine {
    use {
        Breaker b1, b2;
        Protection p;
    }
    link {
        p.O1 -> b1.I;
        b1.O -> p.I1;
        p.O2 -> b2.I;
        b2.O -> p.I2;
    }
    input {
        Ileft = p.I3;
        Iright = p.I4;
    }
    output {
        Oleft = p.O3;
        Oright = p.O4;
    }
}
```

Figure 13. Domain unit specified in *DELTA*.

(the two states being separated by an arrow) and, finally, by the set of output events. The same pattern applies to the specification of *Protection*.

The third clause defines a cluster of automata, called *ProtectedLine*. The *use* section maps a set of automata to a set of names: *Breaker* is mapped to $b1$ and $b2$, while *Protection* is mapped to p. These automata are connected together as specified in the *link* section. Note how cluster *ProtectedLine* is isomorphic to the network displayed in Figure 3. The *input* and *output* sections give names to the dangling terminals of the protection. This allows the cluster to be viewed as a 'structured' communicating automaton defined in terms of 'atomic' communicating automata. Interestingly, clusters can also be defined in terms of other clusters. This results in a hierarchical structure of (possibly structured) communicating automata.

The clusterization mechanism offered by *DELTA* supports reuse in the specification of complex automaton networks, where the same pattern is possibly replicated at several levels of abstractions. With reference to our example, a sequence of protected lines can be defined by aggregating several instances of cluster *ProtectedLine* into a new, higher-level cluster.

Once domain *PowerTransmission* has been defined, the actual automaton network in Figure 3 can be created as an instance of cluster *ProtectedLine*, as follows:

```
use model PowerTransmission;

define system Net: ProtectedLine {}
```

The *use model* clause sets the domain for the instantiation, which is carried out by the subsequent *define system* statement. The latter creates the automaton network *Net* as an instance of *ProtectedLine*. Although it is not the case in our example, *DELTA* allows the specification of a default viewer and/or a default ruler and/or a default initial state within the curly brackets after the define statement: this would require the omission of such pieces of information in the specification of problems relevant to *Net*.

The third class of *DELTA* specification concerns the definition of diagnostic problems. The substantiation in *DELTA* syntax of the diagnostic problem $\wp(Net)$ defined in Example 8 is outlined in Figure 14. The first *use system* clause establishes the context of the problem, namely system *Net*. The actual problem is specified by the *define problem* statement. In our example, the problem (named *PNet*) is defined in terms of initial state, viewer, observation and ruler. Of these, only the *observation* section is mandatory, the others being possibly instantiated by the defaults in the *domain* and *system* units. In our example, the *state* section is actually redundant, as it corresponds to the default initial states for *Breaker* and *Protection*. Note how *viewer* and *ruler* sections implement viewer $\mathcal{V}_N$ and ruler $\mathcal{R}_N$ defined in Tables I and II, respectively.

Similarly, the *observation* section implements the observation $\mathcal{O}_N$ displayed in Figure 5. Specifically, nodes of the observation are named $N1 \cdots N4$. An arrow (resembling the precedence relationship) separates each node from the list of its successive nodes in the observation graph: $N1$ precedes $N2$ and $N3$, these preceding $N4$ on their turn. For leaf nodes ($N4$), the list is empty. Then, the logical content of each node is detailed, where the *null* keyword stands for the empty label ε.

```
use system Net;

define problem PNet {
    state {
        start 0 b1;
        start 0 b2;
        start 0 p;
    }
    viewer {
        o1 @ T1(b1);
        o2 @ T1(b2);
        l @ T2(p);
        r @ T3(p);
    }
    observation {
        N1 -> (N2, N3): (l, null);
        N2 -> (N4): (o1, o2);
        N3 -> (N4): (o1, null);
        N4 -> (): (o1, o2, null);
    }
    ruler {
        s @ T1(p);
        f1 @ T2(b1);
        f2 @ T2(b2);
    }
}
```

Figure 14. Problem unit specified in *DELTA*.

7. DIAGNOSTIC ALGORITHMS

The final goal of the diagnostic environment is generating the solution to diagnostic problems. According to our definition, solving a diagnostic problem $\wp(\aleph) = (\aleph_0, \mathcal{V}, \mathcal{O}, \mathcal{R}, \mathcal{K})$ means yielding the relevant set of candidate diagnoses. According to Equation (10), the solution $\Delta(\wp(\aleph))$ only depends on $\aleph_0$, $\mathcal{V}$, $\mathcal{O}$, $\mathcal{R}$ and the basic subset of $\mathcal{K}$ describing $\aleph$. In other words, the solution does not depend on the compiled knowledge in $\mathcal{K}$, which is only a means of speeding up the diagnostic engine, by avoiding low-level model-based reasoning. However, from a practical point of view, what matters is not only the soundness and completeness of the solution: it is crucial that such a solution be provided within (possibly stringent) time constraints. This is why the role of compiled knowledge is essential in diagnostic-problem solving.

It should be noted that if a diagnosis task is *a posteriori*, it does not mean *per se* that such a task is performed off-line. On the contrary, our approach deals with *a posteriori* diagnosis on-line, that is, while the system is still operating, and the diagnostic result has to be provided under application-dependent time constraints, so as to make a decision that can affect its evolution. Considering our reference example of power networks, as pointed out in Example 1, the operator is supposed to be in a control room to analyze the (possibly large) set of observations generated by a complete reaction of the protection apparatus in order to avoid a black-out. Typically, the decision on which possible repair actions are expected to be made must be performed within one minute. This means that, ideally, the diagnostic system is required to generate the candidate diagnoses in a few seconds. As such, *a posteriori* diagnosis is not a synonym of off-line diagnosis: rather, it is in contrast with *monitoring-based diagnosis*, where a diagnosis is expected to be generated whenever a new piece of observation is perceived [7].

Compiled knowledge can be either *special purpose* or *general purpose*. Special-purpose knowledge consists of the graphs (behavior, abduction, map) generated on-line by the diagnostic engine during the solution of a previous problem. In contrast, general-purpose knowledge is generated off-line, by the preprocessor, and does not depend on a particular observation (although it may depend on a viewer and/or a ruler).

7.1. Solving crude problems

When no compiled knowledge is available, solving a diagnostic problem $\wp(\aleph) = (\aleph_0, \mathcal{V}, \mathcal{O}, \mathcal{R}, \mathcal{K})$ amounts to making up the relevant abduction. According to the informal definition in Section 3.6, an abduction is a graph where each path from the root to a final node is a history of $\aleph$ consistent with the observation. In addition, each node in the abduction embodies a diagnosis encompassing the labels associated with the faulty transitions involved in the history (see Figure 7).

Algorithm 1 (Abduction). *The abduction is generated starting from the initial state of $\aleph$ and connecting new states that are reachable via triggerable transitions. Specifically, a node N in the search space is identified by four fields, namely $N = (\mathbb{A}, \mathbb{L}, \Im, \delta)$, where:*

- *$\mathbb{A} = (\bar{A}_1, \ldots, \bar{A}_n)$ is the record of states of the automata in $\aleph$, where each $\bar{A}_i$, $i \in [1 .. n]$, is a state relevant to an automaton A_i in $\aleph$ (n is the number of automata in $\aleph$);*
- *$\mathbb{L} = (\bar{L}_1, \ldots, \bar{L}_m)$ is the record of states of the links in $\aleph$ (m is the number of links in $\aleph$);*
- *$\Im$ is a node of the index space IndSpace($\mathcal{O}$);*
- *$\delta = \{\varphi_1, \ldots, \varphi_p\}$ is a (possibly empty) set of labels defined in $\mathcal{R}$.*

Node N is final when $\Im$ is final in IndSpace($\mathcal{O}$) and all links in $\mathbb{L}$ are empty. The search starts at the initial node $N_0 = (\mathbb{A}_0, \mathbb{L}_0, \Im_0, \delta_0)$ where $\mathbb{A}_0 = \aleph_0$, $\mathbb{L}_0 = (\emptyset, \ldots, \emptyset)$, $\Im_0$ is the root of IndSpace($\mathcal{O}$), and $\delta_0 = \emptyset$.

Each successor of a given node is obtained by applying an automaton transition which is consistent with both the topology of $\aleph$ and the observation $\mathcal{O}$, based on viewer $\mathcal{V}$. An applied transition is an edge of the search space. Nodes and edges are stored in variables $\mathcal{N}$ and $\mathcal{E}$, respectively. The generation of the abduction is carried out by the following six steps.

(1) *$\mathcal{N} = \{N_0\}$; $\mathcal{E} = \emptyset$.*
(2) *Repeat steps* (3)–(5) *until all nodes in $\mathcal{N}$ are marked.*
(3) *Get an unmarked node $N = (\mathbb{A}, \mathbb{L}, \Im, \delta)$ in $\mathcal{N}$.*
(4) *For each automaton A_i in $\aleph$, for each triggerable transition T in A_i, if T is silent or the label ℓ associated with T in $\mathcal{V}$ marks an edge from $\Im$ to $\Im^*$ in IndSpace($\mathcal{O}$), then perform the following steps:*

(a) *create a node $N' = (\mathbb{A}', \mathbb{L}', \Im', \delta')$ as a copy of N;*
(b) *set $\bar{A}_i$ to the state of A_i reached by T;*
(c) *if the event e triggering T is relevant to a link L_j in $\aleph$, then remove e from $\bar{L}_j$*[††]*;*
(d) *insert the internal output events of T (if any) into the relevant links in $\mathbb{L}'$;*
(e) *if T is observable, then set $\Im'$ to $\Im^*$;*
(f) *if T is faulty, where φ is the relevant label in $\mathcal{R}$, then insert φ into δ';*
(g) *if $N' \notin \mathcal{N}$, then insert N' into $\mathcal{N}$;*
(h) *insert edge $N \xrightarrow{T} N'$ into $\mathcal{E}$;*

(5) *Mark N.*
(6) *Remove from $\mathcal{N}$ all of the nodes and from $\mathcal{E}$ all of the edges, which are not on a path from the initial state N_0 to a final state in $\mathcal{N}$.*

Example 16. Consider the diagnostic problem $\wp(Net) = (Net_0, \mathcal{V}_N, \mathcal{O}_N, \mathcal{R}_N, \mathcal{K}_N)$ defined in Example 8. The relevant abduction $Abd(\wp(Net))$ is shown in Figure 15. The dashed part of the graph involves nodes and edges that do not reach any final node (see step (6) of Algorithm 1). According to the definition, each node is identified by a tuple $N = (\mathbb{A}, \mathbb{L}, \Im, \delta)$: for instance, considering $N = \alpha_7$, we have $\mathbb{A} = (0, 1, 0)$, $\mathbb{L} = (\langle z \rangle, \langle \rangle, \langle \rangle, \langle f \rangle)$, $\Im = \Im_0$ and $\delta = \{s, f_2\}$. Final nodes are α_3, α_8, α_{13} and α_{14}. The solution of $\wp(Net)$ contains the diagnoses associated with such nodes, namely $\Delta(\wp(Net)) = \{\emptyset, \{s\}, \{s, f_1\}, \{s, f_2\}\}$, which is exactly the set of candidate diagnoses determined in Example 8 based on the definition of the problem solution.

7.2. Solving compiled problems

We now assume a diagnostic problem $\wp(\aleph) = (\aleph_0, \mathcal{V}, \mathcal{O}, \mathcal{R}, \mathcal{K})$ where $\mathcal{K}$ involves some sort of compiled knowledge.When generated off-line as the result of a preprocessing activity, this knowledge is bound to support a wide variety of problems. A basic assumption of knowledge compilation is that each knowledge graph in $\mathcal{K}$ (whether a behavior, an abduction or a map) is not constrained by any specific observation.

The most general knowledge graph is the behavior space $Bhv(\aleph, \aleph_0)$, as defined in Section 2. In fact, $Bhv(\aleph, \aleph_0)$ is independent of any viewer or ruler. On the other hand, this generality does not allow for a direct solution of a given $\wp(\aleph)$. In order to solve $\wp(\aleph)$, we need to make up the behavior $Bhv(\wp(\aleph))$ constrained by observation $\mathcal{O}$ and viewer $\mathcal{V}$ and, then, the abduction $Abd(\wp(\aleph))$ inherent to ruler $\mathcal{R}$.

A more constrained knowledge graph is an *abduction space*, that is, an abduction relevant to a behavior space $Bhv(\aleph, \aleph_0)$ and a ruler $\mathcal{R}$, namely $Abd(\aleph, \aleph_0, \mathcal{R})$. Several abduction spaces may be defined for the same behavior space, specifically one for each different ruler. The construction of an abduction space can be carried out based on $Bhv(\aleph, \aleph_0)$ by a straightforward modification of the algorithm defined in Section 7.1. As such, $Abd(\aleph, \aleph_0, \mathcal{R})$ incorporates all possible histories of system $\aleph$ rooted in $\aleph_0$, where each node is decorated with the relevant candidate diagnoses.

[††]No action is performed when e is coming from the outside of $\aleph$, for example, from the standard input or from a link connected to a dangling terminal of $\aleph$.

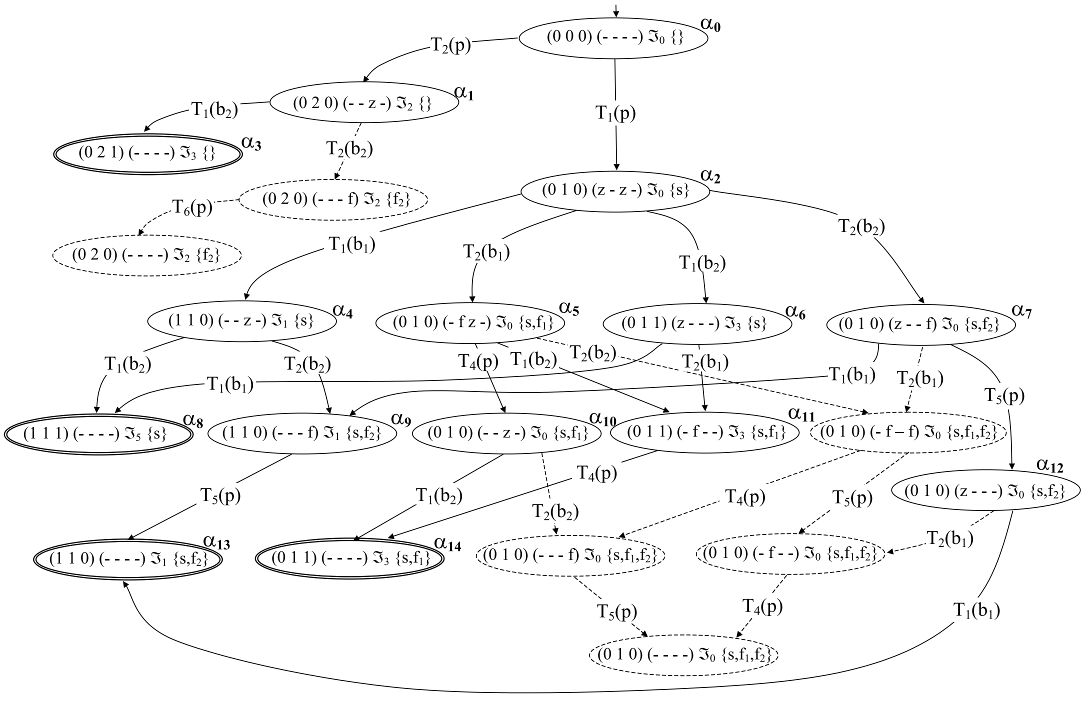

Figure 15. Construction of abduction $Abd(\wp(Net))$.

Example 17. The abduction space $Abd(Net, Net_0, \mathcal{R}_N)$ relevant to the behavior space outlined in Figure 4 and the ruler $\mathcal{R}_N$ specified in Table II is shown in Figure 16. Note how each node of the abduction space is a pair (β, δ), where β is a state of $Bhv(\aleph, \aleph_0)$ and δ a diagnosis based on $\mathcal{R}_N$.

The abduction-space independence of any viewer makes it a valuable diagnostic knowledge for problems of the kind $\wp(\aleph) = (\aleph_0, \mathcal{V}, \mathcal{O}, \mathcal{R}, \mathcal{K})$, where $\aleph_0$ and $\mathcal{R}$ are matched by $Abd(\aleph, \aleph_0, \mathcal{R})$. However, such an independence is paid in terms of potential inefficiency in the solution of $\wp(\aleph)$. Roughly speaking, solving $\wp(\aleph)$ requires generating a restriction of $Abd(\aleph, \aleph_0, \mathcal{R})$ based on the given observation $\mathcal{O}$ and viewer $\mathcal{V}$, and, then, collecting the diagnoses.

However, the drawback of using an abduction space for solving a problem lies in the generic nature of the former, specifically, its independence of viewers. Such an independence prevents the graph from being focused on observable labels, since these need a specific viewer $\mathcal{V}$. If $\mathcal{V}$ were known, the abduction space might be simplified so as to reduce the solution of a diagnostic problem to a pure pattern-matching task, as expected by requirement (4) in Section 4. This consideration leads us to the notion of a map space.

Algorithm 2 (Map space). *Let $Abd(\aleph, \aleph_0, \mathcal{R})$ be an abduction space and $\mathcal{V}$ a viewer for $\aleph$. The map space $Map(\aleph, \aleph_0, \mathcal{V}, \mathcal{R})$ is determined by the following steps:*

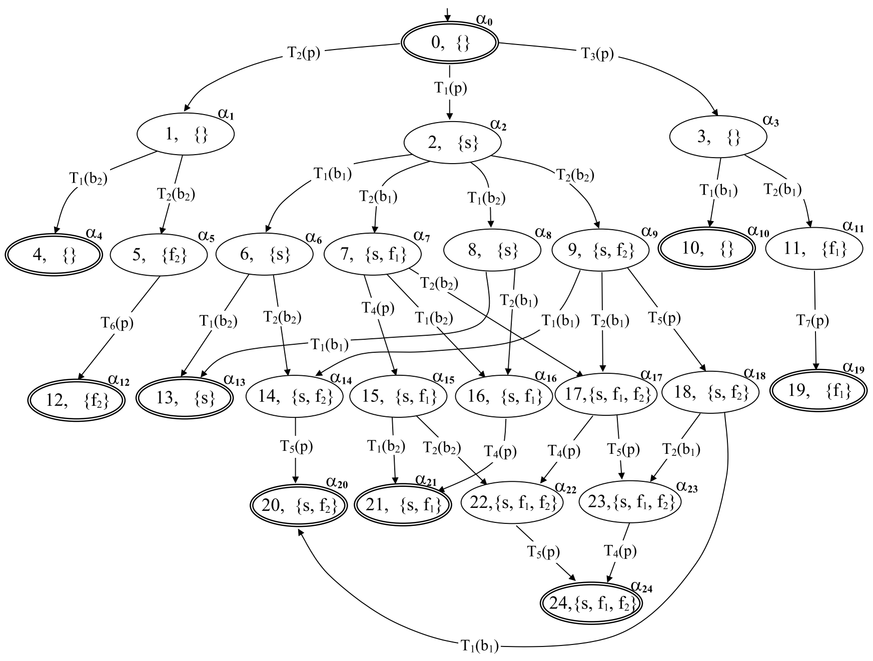

Figure 16. Abduction space $Abd(Net, Net_0, \mathcal{R}_N)$.

(1) *replace the label T marking each edge of $Abd(\aleph, \aleph_0, \mathcal{R})$ with the label ℓ (possibly ε) associated with T in viewer $\mathcal{V}$ (this results in a non-deterministic automaton due to ε-transitions);*

(2) *generate the deterministic automaton equivalent to the non-deterministic automaton yielded in step* (1) *(this results in each state of the deterministic automaton being identified by a subset of the states in the non-deterministic automaton [29]);*

(3) *mark each state S of the deterministic automaton yielded in step* (2) *with the set of candidate diagnoses associated with the final states of the abduction involved in S.*

Example 18. Consider the abduction space $Abd(Net, Net_0, \mathcal{R}_N)$ in Figure 16. Assume viewer $\mathcal{V}_N$ in Table I. In Figure 17(a) the non-deterministic automaton obtained from $Abd(Net, Net_0, \mathcal{R}_N)$ through step (1) is shown, where unlabeled edges are implicitly marked by ε. The actual map space $Map(Net, Net_0, \mathcal{V}_N, \mathcal{R}_N)$ is displayed in Figure 17(b). As such, each node of the map space is marked by a pair $(\mathcal{A}, \mathcal{D})$, where $\mathcal{A}$ is a subset of the nodes of the abduction space (generated according to the subset construction algorithm for transforming the non-deterministic automaton into an equivalent deterministic automaton [29]), while $\mathcal{D}$ is the set of candidate diagnoses relevant to nodes in $\mathcal{A}$. For instance, considering μ_3, we have $\mathcal{A} = \{\alpha_8, \alpha_{16}, \alpha_{21}\}$ and $\mathcal{D} = \{\{s, f_1\}\}$, the latter including only one candidate. Incidentally, all states are final, since they involve at least one final state in $Abd(Net, Net_0, \mathcal{R}_N)$.

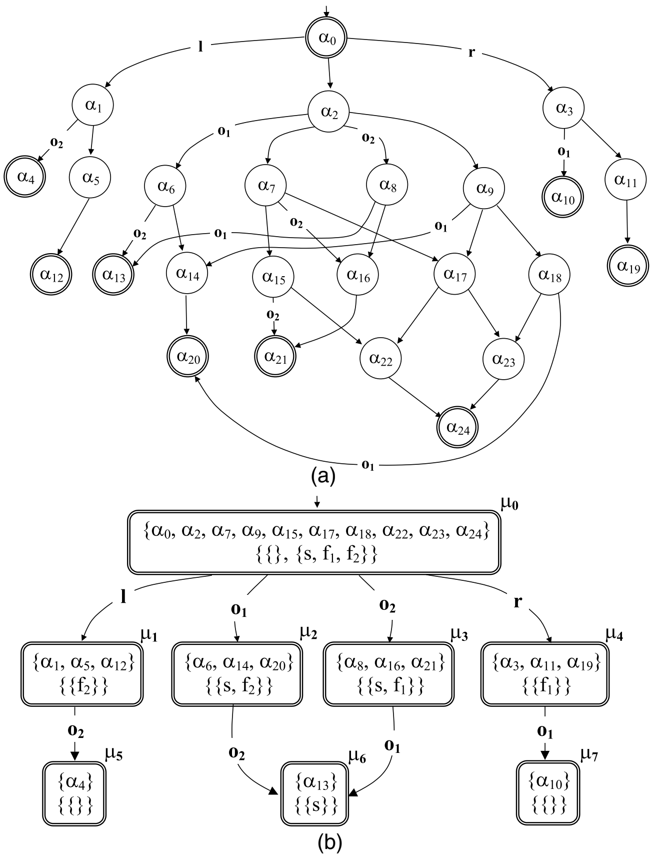
(a)
(b)
μ_0
$\{\alpha_0, \alpha_2, \alpha_7, \alpha_9, \alpha_{15}, \alpha_{17}, \alpha_{18}, \alpha_{22}, \alpha_{23}, \alpha_{24}\}$ $\{\{\}, \{s, f_1, f_2\}\}$
μ_1 $\{\alpha_1, \alpha_5, \alpha_{12}\}$ $\{\{f_2\}\}$
μ_2 $\{\alpha_6, \alpha_{14}, \alpha_{20}\}$ $\{\{s, f_2\}\}$
μ_3 $\{\alpha_8, \alpha_{16}, \alpha_{21}\}$ $\{\{s, f_1\}\}$
μ_4 $\{\alpha_3, \alpha_{11}, \alpha_{19}\}$ $\{\{f_1\}\}$
μ_5 $\{\alpha_4\}$ $\{\{\}\}$
μ_6 $\{\alpha_{13}\}$ $\{\{s\}\}$
μ_7 $\{\alpha_{10}\}$ $\{\{\}\}$
l
r
o_1
o_2

Figure 17. Map space $Map(Net, Net_0, \mathcal{V}_N, \mathcal{R}_N)$ (b) and relevant non-deterministic automaton (a).

When solving a problem, the diagnostic engine first interacts with the knowledge manager to look for relevant compiled knowledge, whether general purpose or special purpose. Specific constraints are to be checked on the compiled knowledge in order to be compatible with the problem to be solved. If a compatible knowledge graph is found, the problem is solved based on such a graph, otherwise crude solving is carried out. Compiled knowledge supports problem solving in different degrees of performance. Roughly speaking, maps are better than abductions and abductions are better than behaviors. Thus, the knowledge manager first looks for maps, then for abductions and, in the end, for behaviors.

Algorithm 3 (Search for reuse). *Assume a problem $\wp(\aleph) = (\aleph_0, \mathcal{V}, \mathcal{O}, \mathcal{R}, \mathcal{K})$, where $\mathcal{K}$ involves some sort of compiled knowledge. Each compiled graph γ belonging to $\mathcal{K}$ solves a problem $\wp'(\aleph) = (\aleph'_0, \mathcal{V}', \mathcal{O}', \mathcal{R}', \mathcal{K}')$, where $\mathcal{K}'$ is undefined, $\mathcal{O}'$ is undefined if γ is a space and $\mathcal{R}'$ is undefined if*

γ is a behavior space (in this context, 'undefined' means 'not constrained', that is, an undefined entity may assume any value of its domain; for instance, $\mathcal{O}'$ is undefined if γ is a space since a behavior/abduction/map space is independent of the observation). For a generic compiled graph γ belonging to $\mathcal{K}$, let:

- *initial_state(γ) be the system state contained in the initial state of γ, that is, $\aleph'_0$;*
- *type(γ) be a univocal identifier of the type of γ (1 for map space, 2 for map, 3 for abduction space, 4 for abduction, 5 for behavior space, 6 for behavior);*
- *length(γ) be the length of IndSpace($\mathcal{O}'$) (that is, the length of its longest path), if $\mathcal{O}'$ is defined,*
- *num_nodes(γ) be the number of nodes of IndSpace($\mathcal{O}'$), if $\mathcal{O}'$ is defined.*

Let subsume(ζ_1, ζ_2) be a function returning the value true in the case where the former parameter subsumes the latter, false otherwise. Both parameters may be viewers, rulers or index spaces.

An algorithm for finding a graph contained in $\mathcal{K}$ that can be used to solve $\wp(\aleph)$ (or returning null in the case where no such graph exists) is described by the following steps.

```
(1)  Extract into $\Gamma$ all of the graphs $\gamma$ in $\mathcal{K}$ such that initial_state($\gamma$) = $\aleph_0$.
(2)  Order the content of $\Gamma$ based on three keys: the order is increasing for the first key (type),
     decreasing for the second key (length) and decreasing for the third key (num_nodes);
(3)  Repeat steps (4)–(14) for each graph $\gamma$ in $\Gamma$, considering such graphs according to their
     order (until a reusable graph is found).
(4)    if type($\gamma$) $\in \{1, 2\}$ then
(5)      if subsume($\mathcal{R}'$, $\mathcal{R}$) and subsume($\mathcal{V}'$, $\mathcal{V}$) then
(6)        if type($\gamma$) = 1 then return $\gamma$
(7)        elsif subsume(IndSpace($\mathcal{O}'$), IndSpace($\mathcal{O}$)) then return $\gamma$
(8)    if type($\gamma$) $\in \{3, 4\}$ then
(9)      if subsume($\mathcal{R}'$, $\mathcal{R}$) then
(10)       if type($\gamma$) = 3 then return $\gamma$
(11)       elsif subsume($\mathcal{V}'$, $\mathcal{V}$) and subsume(IndSpace($\mathcal{O}'$), IndSpace($\mathcal{O}$)) then return $\gamma$
(12)   if type($\gamma$) $\in \{5, 6\}$ then
(13)     if type($\gamma$) = 5 then return $\gamma$
(14)     elsif subsume($\mathcal{V}'$, $\mathcal{V}$) and subsume(IndSpace($\mathcal{O}'$), IndSpace($\mathcal{O}$)) then return $\gamma$
(15) return null.
```

Note that, as will be briefly discussed in Section 10, Algorithm 3 embodies just one of the possible (heuristic) strategies that can be used to extract a reusable chunk of knowledge from the knowledge base, specifically that which we have adopted in our diagnostic environment. The rationale behind the present strategy is that the reusable chunk is searched first within maps, then within abductions and, finally, within behaviors, since the time needed for extracting the solution of the new problem is shorter if the chunk is a map, longer if it is an abduction and even longer if it is a behavior. Within the set of all of the available graphs of the same type (map or abduction or behavior), each graph is considered in reverse order with respect to its extension, since, the larger the extension, the more likely that the current problem is subsumed. In particular, the space graph is considered before any observation-dependent graph since its coverage is larger. The second and third keys for establishing the order according to which the graphs within the knowledge base have to be searched give an estimate of each graph extension (we have assumed that, given two distinct observation-dependent graphs of the same type, whose length of the index space is the same, that having the greater number of index-space nodes has the larger extension).

Once a knowledge graph γ has been selected, problem solving is performed as a matching between γ and the observation $\mathcal{O}$. From an abstract point of view, the matching operation does not depend on the nature of γ: a new graph μ is generated as a refinement of γ, where each path in μ is also a path in γ that matches $\mathcal{O}$. In other terms, the extension of μ is the subset of the extension of γ that is consistent with $\mathcal{O}$. A node of μ is identified by a pair $(\mathcal{G}, \Im)$, where $\mathcal{G}$ is a node of γ and $\Im$ a node of $\mathit{IndSpace}(\mathcal{O})$. On the other hand, the way the solution is actually distilled from μ varies according to the nature of γ. Specifically, if γ is a behavior, we need to generate the relevant abduction based on μ. Instead, if γ is an abduction, each set of fault labels associated with a final node is a candidate diagnosis. Finally, if γ is a map, the solution is the union of all of the sets of diagnoses associated with final nodes.

Algorithm 4 (Matching graph). *Assume a map γ. Let $N_0 = (\gamma_0, \Im_0)$ be the root of the matching graph μ, where γ_0 and $\Im_0$ are the roots of γ and IndSpace($\mathcal{O}$), respectively. Let $\mathcal{N}$ and $\mathcal{E}$ denote the set of nodes and the set of edges of $\mathcal{N}$, respectively. The construction of μ is based on the following steps.*

```
(1) $\mathcal{N} = \{N_0\}$; $\mathcal{E} = \emptyset$.
(2) Repeat steps (3)–(5) until all nodes in $\mathcal{N}$ are marked.
(3) Get an unmarked node $N = (\mathcal{G}, \Im)$ in $\mathcal{N}$.
(4) For each edge $\mathcal{G} \xrightarrow{\ell} \mathcal{G}'$ in $\gamma$, such that $\Im \xrightarrow{\ell} \Im'$ is an edge in IndSpace($\mathcal{O}$), do the following:
    (a) create a node $N' = (\mathcal{G}', \Im')$;
    (b) if $N' \notin \mathcal{N}$, then insert $N'$ into $\mathcal{N}$;
    (c) insert edge $N \xrightarrow{\ell} N'$ into $\mathcal{E}$.
(5) Mark $N$.
(6) Remove from $\mathcal{N}$ all of the nodes and from $\mathcal{E}$ all of the edges that are not on a path from $N_0$
    to a final node $N_f = (\mathcal{G}_f, \Im_f) \in \mathcal{N}$, where $\mathcal{G}_f$ and $\Im_f$ are final nodes in $\gamma$ and IndSpace($\mathcal{O}$),
    respectively.
```

Example 19. Consider the diagnostic problem $\wp(\mathit{Net})$ defined in Example 8, and crudely solved in Example 16. Assume the availability of the map space $\mathit{Map}(\mathit{Net}, \mathit{Net}_0, \mathcal{V}_N, \mathcal{R}_N)$ (Figure 17) in the knowledge base. Accordingly, the diagnostic engine can yield the solution of $\wp(\mathit{Net})$ by matching $\mathit{Map}(\mathit{Net}, \mathit{Net}_0, \mathcal{V}_N, \mathcal{R}_N)$ with $\mathcal{O}_N$, as detailed in Algorithm 4. The resulting matching graph is shown in Figure 18(c). The map space outlined in Figure 17 (node details are omitted) and $\mathit{IndSpace}(\mathcal{O}_N)$ (see Figure 6) are shown in Figures 18(a) and (b), respectively. The actual solution $\Delta(\wp(\mathit{Net}))$ is obtained by aggregating the sets of diagnoses associated with the final nodes, namely $\Delta(\wp(\mathit{Net})) = \{\emptyset, \{s\}, \{s, f_1\}, \{s, f_2\}\}$. As expected, this is equal to the solution found in Example 16.

8. IMPLEMENTATION

Once a diagnostic problem, which is inherent to a system, has been given as input to the diagnostic engine, the latter sends the name of the system to the knowledge manager module, which, in turn, checks whether such a system is known (i.e. its *DELTA* description has already been compiled) or not.

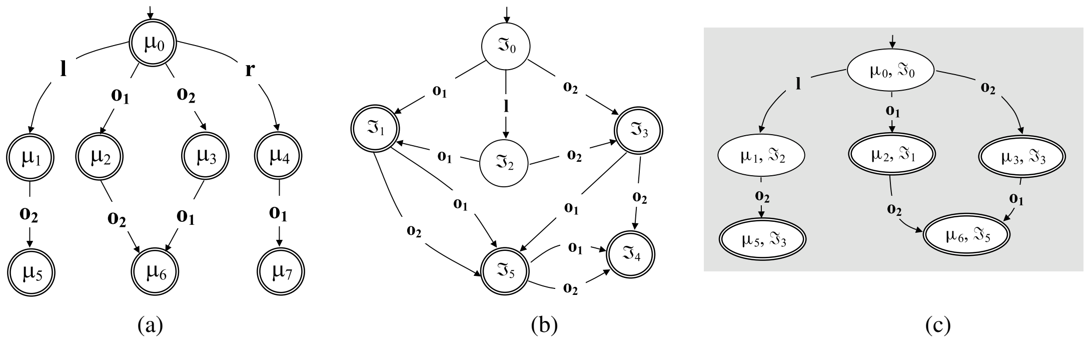

Figure 18. Matching graph (c) resulting from the matching of the map space (a) and the index space (b).

(1) In the case where the system is known to the knowledge manager, the compilation of the problem description has to be performed by the compiler. Once the compilation process has been successfully completed, the knowledge manager checks whether the current problem is subsumed by an already solved problem inherent to the same system (including problems with unknown observations, i.e. behavior/abduction/map spaces). Performing this check means repeatedly matching the features of the current problem (initial state, ruler, viewer and observation) against those of already solved problems. As hinted in Section 7.2, such comparisons are accomplished according to a priority order. The check terminates as soon as a fitting chunk of knowledge has been identified. The reusable knowledge (i.e. the pathname of the file wherein such knowledge is contained) is passed to the diagnostic engine, which exploits it for solving the current problem. If no reusable chunk of knowledge can be identified, the knowledge manager notifies to the diagnostic engine that the current problem is a crude problem, and the diagnostic engine solves it in the appropriate way. After the problem has been solved, the diagnostic engine sends the (reference to the) files it has produced to the knowledge manager, which assigns them proper names, according to homogeneous naming rules, saves them into the right directories and creates the database entries for representing the correspondence between them and the solved problem.

(2) In the case where the current system is unknown to the knowledge manager module, the problem cannot be solved. This impossibility is notified to the diagnostic engine, which displays a warning message to the user, explaining that no such system exists. In order to solve the problem, the compilation of the system instantiation (file *.sys*) is required and, before that, the compilation of the domain (file *.model*) that includes the description of the cluster of which the system is an instance. New descriptions coded in *DELTA* are processed by the compiler, which performs syntax and semantic analysis, generating error and warning messages appropriately. If the description of the domain is correct, it is sent to the knowledge manager, to be permanently stored in the File System. While compiling a system description, the compiler is supposed to retrieve the models of its components, clusters and links from the domain description

the system refers to. The compiled version of the system is sent to the knowledge manager, to be saved in the database.

The activities described above require the knowledge manager to exchange information to and from the compiler and the diagnostic engine as well. Moreover, a bidirectional flow of information between the knowledge manager module and the preprocessor is also needed, since it is assumed that preprocessing commands for generating general-purpose knowledge are issued by the knowledge manager and, once they have been fulfilled, the preprocessor sends its output to the knowledge manager, quite similarly to what is done by the diagnostic engine when generating special-purpose knowledge during on-line processing.

8.1. User categories

Four categories of actors are supposed to interact with the diagnostic environment: *diagnostician*, *domain modeler*, *system modeler* and *knowledge supervisor*. The diagnostician is actually the final user of a diagnostic environment, the user category for which such an environment is developed, that is, everybody supplying a diagnostic problem. Note that the diagnostician may range from a qualified operator, allowed to describe all of the parts of a diagnostic problem, down to a person using the environment as a black-box work tool for deriving diagnoses, possibly ignoring how a discrete-event system is described and having no insight into the notions of a ruler and a viewer. In this case, the diagnostician just inputs the current observation of the system instance to be diagnosed, while the initial state, the ruler and the viewer are implicitly assigned default values.

The other categories are meant to manage and maintain the content of the knowledge base over time. Specifically, a domain modeler is someone authorized to create and compile domain models, where such models are of general validity, that is, they can be exploited several times, both for generating general-purpose knowledge and for solving specific problems inherent to distinct system instances. Therefore, a modeler has to be an expert of the application domain that they are assumed to describe. The domain modeler has to use *DELTA*, however they need to know distinct, more complex, features of the language with respect to the diagnostician, typically they have to know the primitives to describe components, clusters and links. In the description of each component model, the specification of the default initial state has to be included, this being the only possible initial state of the component at the beginning of whichever reaction, if this is the case, or the most common initial state.

A system modeler is someone authorized to create a system, as an instance of a cluster in an existing domain model and to properly characterize the system defaults, i.e. initial state, ruler and viewer. This category of users is distinct from the previous category since, while the domain modeler is an expert of a certain application domain and, typically, of a certain class of devices, they are not necessarily experts of the context wherein a specific device instance is operating, this being what a system modeler is assumed to be. In fact, the initial state of the system may depend on the working context, its viewer depends on the characteristics of the channels conveying the observable labels from the system to the observer and on the sensitivity of such an observer, while the ruler depends on which kinds of phenomena it is necessary/interesting to detect in the current context. In order to generate systems, the system modeler is allowed to consult the *DELTA* descriptions of existing domains (but they cannot change them). The initial state value provided in the system description overwrites the default initial state values given in the domain. Note that the descriptions provided by the system modeler are constraining for the diagnostician, in that the latter cannot ask for the solution of any diagnostic problem inherent to a system that has not been instantiated by the former, and, in the case where the former has supplied default values for a system instance, the latter cannot overwrite such values in any problem description inherent to the same instance.

The knowledge supervisor is in charge of general-purpose knowledge generation and compiled-knowledge maintenance. For instance, a knowledge supervisor can ask the knowledge manager for preprocessing actions inherent to a given system. To fulfill these requests, the knowledge manager module forwards the commands to the preprocessor, then receives the generated graphs, saves them persistently in the File System and updates the database so as to keep track of and index them. Unlike the other user categories, the knowledge supervisor does not write *DELTA* code (but they can understand it) and they do not need to be an expert in any application domain or local context. Instead, they are expected to know the algorithms and strategies exploited by the diagnostic environment, so as to perform reasonable and advantageous knowledge maintenance actions, by using an *ad hoc* command language. Moreover, the knowledge supervisor is allowed to browse the content of both the database and the knowledge base.

The different responsibilities of the user categories are supported by three distinct interaction modes, as displayed in Figure 12:

- the diagnostician interacts with the diagnostic engine, by supplying problem descriptions;
- the modeler directly interacts with the compiler, by supplying domain and system descriptions;
- the knowledge supervisor interacts with the knowledge manager interface thorough knowledge-related commands (both preprocessing and maintenance commands).

8.2. Data structures

Although the diagnostic engine and the preprocessor are distinct modules in the diagnostic environment (Figure 12), they share the same algorithms and the same data structures (actually no data structure for the observation is needed in the preprocessor). Chained data structures have been adopted in the source code of both the diagnostic engine and the preprocessor to handle all graphs. Such data structures mirror the pictorial representations given in this paper to the graphs themselves. Figure 19 displays the C structure for the internal representation of an observation in the diagnostic engine. The *DELTA* observation description in Figure 14, for instance, translates into the graph in Figure 5, that is, into four nodes, each of which is an instance of structure *BhObservation*. The *event* field of the structure is a vector of observable events that constitute the logical content of the relevant node, and *n_event* is the number of such events. So, the value of *n_event* of nodes N1 to N3 is 2 while that of node N4 is 3. The dynamic vectors *child* and *parent* contain the pointers to the child and parent nodes of the current node, respectively, while *n_child* and *n_parent* are the total number of such nodes. So, the value of *n_child* is 2 for N1, 1 for N2 and N3 and 0 for N4. The value of *n_parent*, instead, is 0 for N1, 1 for N2 and N3 and 2 for N4.

The observation graph then has to be transformed into an index space, which is an acyclic deterministic automaton. In the current version of the diagnostic environment, this transformation is performed in two steps: first the observation graph is converted into a non-deterministic automaton, where each edge is labeled by an observable event and each path from the root to a final node is a symptom; then, the non-deterministic automaton is converted into the index space. To this end, a depth-first search of the non-deterministic automaton is performed.

```
typedef struct obs {
    int level;
    char *label;
    BhLabel *event;
    int n_event;
    struct obs **child;
    int n_child;
    struct obs **parent;
    int n_parent;
} BhObservation;
```

Figure 19. C code for a node of the observation graph.

Figure 20 describes the relations between the structures used to represent the knowledge graphs. A node of either a behavior or a behavior space is an instance of the *BhNode* structure. The *idnode* field is the unique (integer) identifier of such a node. The *state* field is the composition of the states of all components. In fact, it is a vector of strings, where each string is the identifier of a component state. The *event* field hosts the content of all links. It is a vector of instances of the *BhStateEvent* structure, each instance being a pair (link identifier, event identifier), where each identifier is an integer. The *child* field represents the list of child nodes of the current node. It is a vector of instances of the *BhChild* structure, each instance being a pair whose former field is the pointer to the child node (which is a *BhNode* object itself) and whose latter field is the pointer to the transition (*BhTransition* object) leading from the current node to its successor. In other words, each behavior node contains the (indirect) pointers to all of its child nodes as well as to all of the transitions that lead to them. Each of these transitions is a *BhTransition* object, where *BhTransition* is a structure used to represent the transitions in component models[‡‡].

The *ind_space* field of *BhNode* is used only when the graph is a behavior, while it is useless if the graph is a behavior space as it is a pointer to the index space state (of the considered observation) corresponding to the sequence of messages generated during the evolution from the initial node to the current node. The remaining four fields of *BhNode* are Boolean. They do no bring any information about the content of the node or its relations; rather, they describe its nature. The *path_to_non_quiescent* is set to true during the behavior construction if there does not exist any path from the current node to a quiescent node and, therefore, the current node was visited by the construction process but it does not belong to the behavior (space), so it as to be discarded once the construction has been completed. The *quiescent* field is true if the current node is quiescent. The last two flags are meaningful only if the graph is a behavior while they are useless if it is a behavior space. The *path_to_solution* field is set to true during the behavior construction if there exists a path at least from the current node to a final node and, therefore, the current node belongs to the behavior. Finally, *solution* is true if the current node is final (i.e. it is both a quiescent node and a node whose linked index space state, pointed by *ind_space*, is final, that is, it corresponds to a complete observation instance).

[‡‡]In Figure 20, the dots stand for further system-modeling structures.

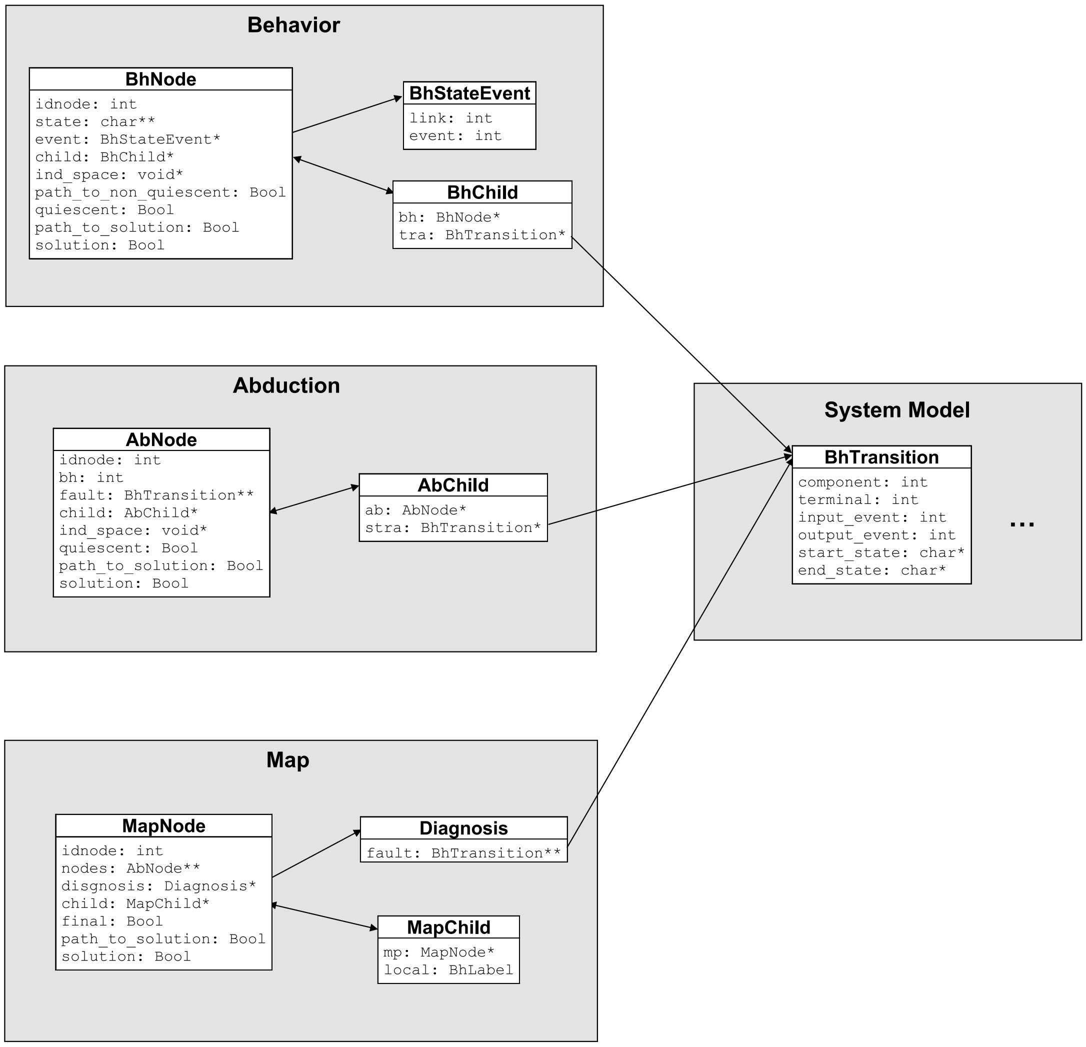

Figure 20. Data structures implementing the knowledge graphs.

The *AbNode* structure, representing a node of either an abduction or an abduction space, is quite similar to *BhNode*. The *bh* field is the (integer) identifier of the behavior node contained in the current abduction node. The *fault* field is the set of faulty transitions, according to the current ruler, corresponding to the path from the initial node to the current node. No *path_to_non_quiescent* field is needed since the abduction (space) is drawn from a behavior (space) and the derivation process does not generate any inconsistent state (or, in other words, it does not need any backtracking step). Like the homonymous field of the *BhNode* structure, *solution* is meaningful only if the graph is an abduction while it is useless if it is an abduction space.

It is now easy to survey the fields of the *MapNode* structure, used to represent each node of either a map or a map space. The *nodes* field is the set of pointers to all of the abduction nodes belonging to the current map node. The *diagnosis* field is a pointer to a *Diagnosis* object, this being a dynamic list, each element of which is a set of faulty transitions. The *final* field signals whether the current node includes any quiescent abduction nodes. The implementation of the parent-to-child relation is the same as in the two previous node structures, however here a edge has no reference to a component transition since it does not correspond to a single component transition, instead it is marked by an observable label, contained in the *label* field of the *MapChild* structure.

9. EXPERIMENTAL RESULTS

The diagnostic environment has been experimented on a variety of diagnostic problems, based on the application domain of power transmission lines introduced in Section 2 and specified in Section 6 by means of the *DELTA* language. Experimental results have concentrated on the cost of network preprocessing and diagnostic-problem solving in terms of space and time.

Network modeling is based on the cluster *ProtectedLine* defined in Figure 13, where a protected line is defined as the composition of a protection and two breakers. Several networks have been defined, each of them corresponding to the concatenation of a different number of lines. In what follows, Net_i denotes the network consisting of i protected lines and, therefore, including $i \cdot 3$ components. Seven classes of experiments have been considered, as detailed in the following.

9.1. First class of experiments

The first class of experiments is composed of 30 crude diagnostic problems, relevant to networks Net_i, $i \in [1 .. 30]$, respectively. In each problem, the observation is linear*, without any uncertainty. Specifically, all breakers are observed to open sequentially from left to right in Net_i. Results are outlined in Table IV, where, for each experiment, the cost of generating the behavior and the subsequent abduction are reported. For the sake of reusability, we have generated the two graphs separately, yielding the abduction based on the behavior. For each graph, three results are given, namely the number of nodes, the number of edges and the time (in seconds) to yield the graph. Note that the numbers of nodes and edges refer to the *search space*, rather than the actual resulting graph. For example, considering Net_{10}, we have 11 206 nodes in the search space of the behavior, but only

*A linear observation is a totally temporally ordered sequence of precise observable labels.

Table IV. Results for the first class of experiments.

	Bhv			*Abd*		
CAN	Nodes	Edges	Time	Nodes	Edges	Time
Net_1	25	31	0.00	4	3	0.00
Net_2	100	133	0.00	7	6	0.00
Net_3	272	377	0.00	10	9	0.00
Net_4	602	857	0.00	13	12	0.00
Net_5	1166	1691	0.01	16	15	0.00
Net_6	2055	3021	0.02	19	18	0.00
Net_7	3375	5013	0.03	22	21	0.00
Net_8	247	7857	0.05	25	24	0.01
Net_9	7807	11 767	0.07	28	27	0.00
Net_{10}	11 206	16 981	0.13	31	30	0.00
Net_{11}	15 610	23 761	0.20	34	33	0.00
Net_{12}	21 200	32 393	0.32	37	36	0.00
Net_{13}	28 172	43 187	0.42	40	39	0.00
Net_{14}	36 737	56 477	0.77	43	42	0.00
Net_{15}	47 121	72 621	1.13	46	45	0.00
Net_{16}	59 565	92 001	1.78	49	48	0.00
Net_{17}	74 325	115 023	2.81	52	51	0.01
Net_{18}	91 672	142 117	3.59	55	54	0.00
Net_{19}	111 892	173 737	5.86	58	57	0.00
Net_{20}	135 286	210 361	7.96	61	60	0.00
Net_{21}	162 170	252 491	14.02	64	63	0.01
Net_{22}	192 875	300 653	25.94	67	66	0.01
Net_{23}	227 747	355 397	40.92	70	69	0.00
Net_{24}	267 147	417 297	66.25	73	72	0.01
Net_{25}	311 451	486 951	76.23	76	75	0.02
Net_{26}	361 050	564 981	73.57	79	78	0.00
Net_{27}	416 350	652 033	124.8	82	81	0.01
Net_{28}	477 772	748 777	111.4	85	84	0.02
Net_{29}	545 752	855 907	204.5	88	87	0.01
Net_{30}	620 741	974 141	258.2	91	90	0.03

31 nodes in the corresponding abduction. This means that the vast majority of the search space of the behavior is composed of *spurious* nodes, which are not connected to any final node of the actual behavior. On the other hand, evidence indicates that, once the behavior is yielded, the relevant abduction is generated much more efficiently than the behavior. In a nutshell, the cost of crude-problem solving stands on the behavior-generation side.

9.2. Second class of experiments

The second class of experiments involves diagnostic problems where the observation is no longer linear. In particular, for network Net_i, the observation is a disconnected graph of i connected subgraphs,

Table V. Results for the second class of experiments.

	Indspace			*Bhv*			*Abd*		
CAN	Nodes	Edges	Time	Nodes	Edges	Time	Nodes	Edges	Time
Net_2	9	13	0.00	158	214	0.00	13	12	0.00
Net_3	27	55	0.00	685	970	0.01	46	45	0.00
Net_4	81	217	0.00	2412	3507	0.04	193	192	0.00
Net_5	243	811	0.02	7439	11 005	0.15	976	975	0.00
Net_6	729	2917	0.12	20 958	31 380	0.54	5869	5868	0.56
Net_7	2187	10 207	0.86	55 353	83 604	2.72	41 098	41 097	17.53

Table VI. Results for the third class of experiments.

	Bhv			*Abd*		
CAN	Nodes	Edges	Time	Nodes	Edges	Time
Net_2	192	279	0.00	140	195	0.00
Net_3	1058	1545	0.02	559	800	0.01
Net_4	4929	7135	0.07	2066	2987	0.01
Net_5	20 801	29 806	0.46	7359	10 684	0.03
Net_6	82 451	117 129	5.47	25 718	37 405	0.11
Net_7	313 385	442 184	81.80	88 963	129 494	0.41

each subgraph being composed of two nodes relevant to the opening of the two breakers (from left to right) of Net_i. In other words, although all breakers in Net_i are observed to open, temporal ordering is only partially defined on pairs of breakers relevant to the same line. Results are reported in Table V, which shows not only the costs for generating behaviors and abductions, but also those for yielding the index space of the observation. Note how, unlike the previous class of experiments, we can no longer neglect the cost of generating the abduction.

9.3. Third class of experiments

In the third class of experiments, it is uncertain whether each breaker has opened or not. In other words, the observation graph of each diagnostic problem is connected and represented as a sequence of nodes, where each node is relevant to a different breaker. However, the content of each node of the observation graph is marked by two labels: the labels indicating the opening of the breaker and the null label ε. Therefore, two extreme cases hold: (1) all breakers opened; and (2) no breakers opened. In between, all combinations are possible, owing to the uncertainty of the observation. Results are reported in Table VI. Note that, as in Table IV, the cost (time) of crude-problem solving stands on the behavior-generation side.

Table VII. Results for the fourth class of experiments.

	Bhv			*Abd*		
CAN	Nodes	Edges	Time	Nodes	Edges	Time
Net_2	140	197	0.00	20	22	0.00
Net_3	454	646	0.01	29	31	0.00
Net_4	1120	1615	0.01	38	40	0.00
Net_5	2336	3410	0.04	47	49	0.00
Net_6	4345	6409	0.09	56	58	0.00
Net_7	7435	11 062	0.13	65	67	0.00
Net_8	11 939	17 891	0.22	74	76	0.00
Net_9	18 235	27 490	0.44	83	85	0.00
Net_{10}	26 746	40 525	0.78	92	94	0.00
Net_{11}	37 940	57 734	1.17	101	103	0.00
Net_{12}	52 330	79 927	2.54	110	112	0.00
Net_{13}	70 474	107 986	4.18	119	121	0.01
Net_{14}	92 975	142 865	7.22	128	130	0.01
Net_{15}	120 481	185 590	11.96	137	139	0.00
Net_{16}	153 685	237 259	19.39	146	148	0.00
Net_{17}	193 325	299 042	29.68	155	157	0.00
Net_{18}	240 184	372 181	45.03	164	166	0.02
Net_{19}	295 090	457 990	67.87	173	175	0.02
Net_{20}	358 916	557 855	95.58	182	184	0.01

9.4. Fourth class of experiments

The fourth class of experiments is somewhat similar to the third, as the observation graph of each diagnostic problem is represented by a sequence of nodes, each node being relevant to a distinct breaker. However, that half of these nodes are marked by just one label, while the other half is marked by two labels, one denoting the opening of the breaker, the other being the null label ε. As such, these observations are less uncertain than those of the previous class, as only half of the nodes in each observation graph are uncertain. Results for these experiments are reported in Table VII. Consider the same network Net_7 in both Tables VI and VII. Note how, due to less uncertainty, the figures in Table VII are considerably smaller than the corresponding figures in Table VI. For example, the number of nodes processed for the behavior in Table VII is 7435, while the corresponding number in Table VI is 313 385. This proportion is even more striking for the computation time (0.13 versus 81.80 s).

9.5. Fifth class of experiments

The fifth class of experiments takes into account observations where only two breakers are observed to open: the left breaker of the first line and the right breaker of the last line. Results are reported in Table VIII. Note how, considering the computation time, the cost of behavior generation is neglectable

Table VIII. Results for the fifth class of experiments.

	Bhv			*Abd*		
CAN	Nodes	Edges	Time	Nodes	Edges	Time
Net_2	71	98	0.00	16	18	0.00
Net_3	150	222	0.00	59	81	0.00
Net_4	271	421	0.00	176	264	0.00
Net_5	443	713	0.01	453	711	0.00
Net_6	675	1116	0.02	1062	1710	0.01
Net_7	976	1648	0.01	2347	3837	0.01
Net_8	1355	2327	0.02	4996	8244	0.03
Net_9	1821	3171	0.03	10 385	17 235	0.12
Net_{10}	2383	4198	0.06	21 266	35 418	0.41
Net_{11}	3050	5426	0.07	43 143	72 009	1.68
Net_{12}	3831	6873	0.08	87 024	145 440	6.54
Net_{13}	4735	8557	0.12	174 925	292 575	26.18
Net_{14}	5771	10 496	0.14	350 878	587 142	106.87

with respect to that of computing the abduction. This resembles qualitatively the results outlined in Table V. For example, with reference to Net_{14}, it is striking the proportion between the number of nodes processed for the behavior (5771) and that for the abduction (350 878). This is also reflected on the computation time (0.14 versus 106.87 s).

9.6. Sixth class of experiments

The sixth class of experiments concentrates on preprocessing rather than on solving specific diagnostic problems. Results are reported in Table IX, where all three sorts of graph are considered, namely behavior space, abduction space and map space. With reference to the behavior and abduction spaces in Table IX, note how, generally speaking, figures are considerably larger than the corresponding figures in the other tables. For example, costs for network Net_7 in Table IX are far heavier than those in the other tables, with the exception of Table VI. In fact, the cost for generating the behavior space for Net_7 is essentially the same as that for generating the behavior of Net_7 based on a particularly uncertain observation (owing to the inclusion of the null label in the nodes of the observation graph). On the other hand, this essential equivalence does not hold for abductions: the cost for generating the abduction space for Net_7 (Table IX) is very larger than that for generating the abduction of the same network in Table VI, specifically 126.11 versus 0.41 s.

9.7. Seventh class of experiments

The seventh class of experiments focuses on solving compiled problems, that is, diagnostic problems where some sort of compiled knowledge can be reused, as discussed in Section 7.2. Experiments have shown that what makes the difference is whether an exploitable abduction is available or not.

Table IX. Results for the sixth class of experiments.

	Bhv			*Abd*			*Map*		
CAN	Nodes	Edges	Time	Nodes	Edges	Time	Nodes	Edges	Time
Net_1	25	34	0.00	29	37	0.00	4	5	0.00
Net_2	192	297	0.00	168	240	0.00	20	31	0.00
Net_3	1058	1731	0.01	912	1340	0.01	107	183	0.00
Net_4	4929	8299	0.08	5044	7428	0.02	617	1105	0.03
Net_5	20 801	35 604	0.49	30 112	44 012	0.18	3926	7209	0.17
Net_6	82 451	142 591	6.80	199 348	288 148	3.47	27 834	51 923	1.92
Net_7	313 385	545 744	84.95	1 475 560	2 107 724	126.11	219 879	414 735	246.07

Specifically, when the abduction (possibly an abduction space) is available, the time for solving the problem is very short (almost null for problems involving networks up to Net_7). Instead, when no abduction can be exploited, the computation time basically stands on the abduction-generation side, with extremely changeable results. However, even in this case, there is a gain in efficiency with respect to solving the same problem crudely. This can be explained by the fact that the search within the abduction is much faster than the creation of the latter. The experimental results for solving 20 different diagnostic problems relevant to the same network Net_{20} (20 lines) are depicted in Figure 21. The x-axis indicates the size of the observation (number of nodes) of each problem. The y-axis indicates the time (in seconds) to solve the problem. Two curves are given, relevant to problem solving with and without reuse, respectively. Note how reusing knowledge graphs makes the computation time of problem solving (including the search for reusable knowledge) almost constant (about one second). For instance, solving the problem with 20 nodes in the observation requires 1.16 versus 56.09 s (without reuse).

9.8. Eighth class of experiments

The eighth class of experiments tests the time needed for searching reusable graphs in the knowledge base. We have considered more than 400 different instantiations of the knowledge base, where each instantiation KB_i, $i \in [1 .. 408]$, contains i different diagnostic problems for the same network, each problem referring to an observation generated randomly. The diagram shown in Figure 22 indicates the time (in seconds) for searching reusable knowledge for each knowledge-base instantiation KB_i. Note how the search time is essentially linear with the size of the knowledge base.

10. DISCUSSION

The implemented diagnostic environment, being a proof-of-concept prototype, suffers from many limitations.

First, in the trade-off between time and space, time was privileged. This results in a heavy RAM allocation since, if a graph-based chunk of knowledge is produced or reused, the whole graph

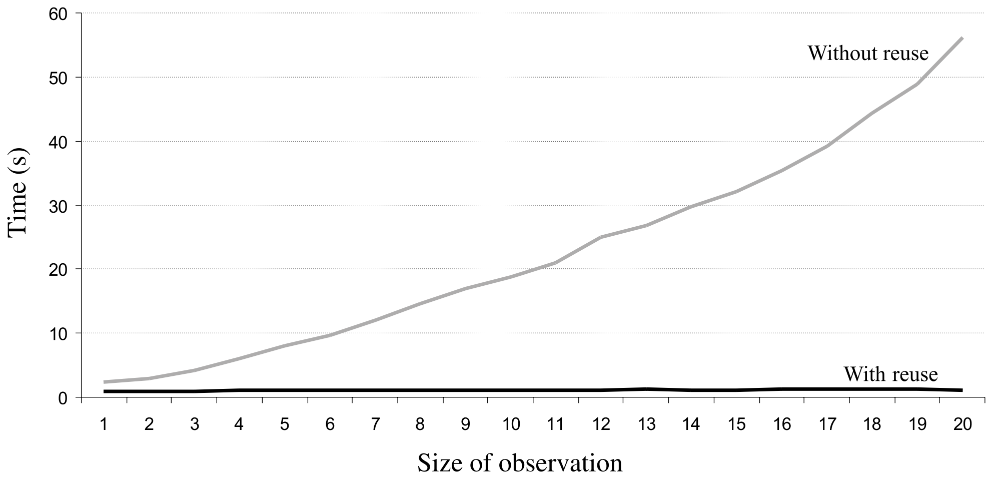

Figure 21. Solving problems with and without reuse.

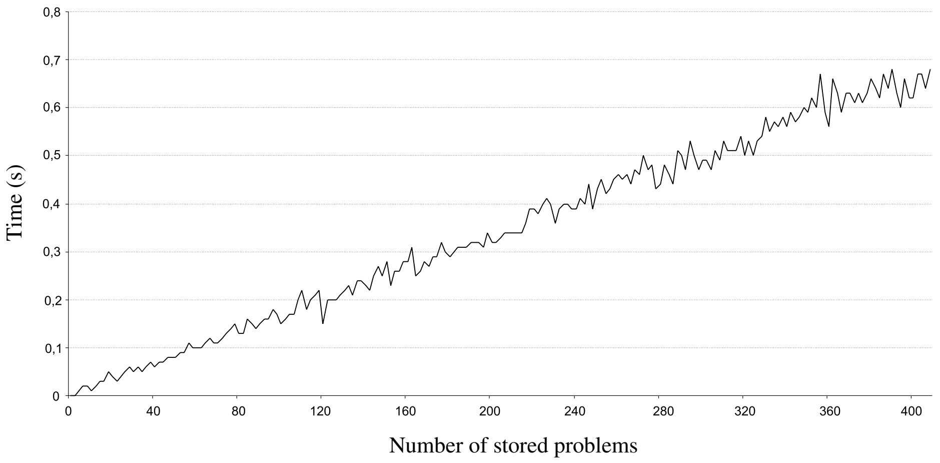

Figure 22. Searching for reusable knowledge.

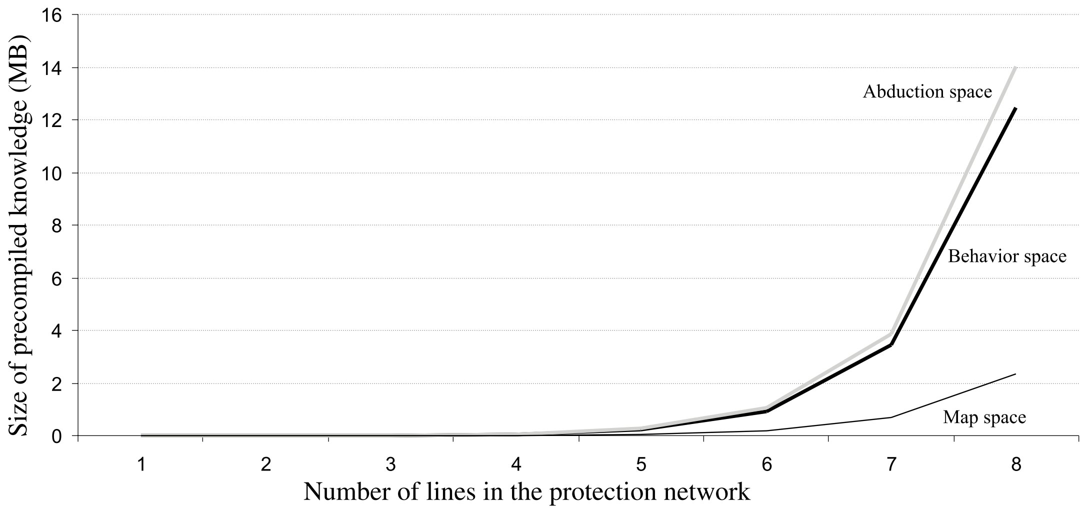

Figure 23. Space allocation.

is handled. Given several graphs sharing the same extension in terms of encompassed histories, abductions are the largest. In the precompilation experiments we performed, the abduction space of a system consisting of eight power transmission lines (24 components altogether) occupied 13 MB. Therefore, the feasibility of experiments was limited by the size of the considered systems, not by the length of the response time. For further details as to the size of precompiled structures (behavior space, abduction space and map space), see Figure 23.

Second, the algorithm adopted for subsumption checking is naive. The experiments performed to record how the time for retrieving a reusable graph (the subsumption check of each candidate graph included) is affected by the number of graphs in the knowledge base showed that the growth is linear with the number of processed graphs referring to the same system. Although all of the performed experiments showed that, given a problem, the CPU time needed to choose a chunk of reusable knowledge is negligible (less that one second with an individual knowledge base of a system including more than 400 graphs), such a time could still be reduced. In fact, in order to establish whether a chunk of knowledge (inherent to both the same system and the same initial state the current problem refers to) can be reused, the subsumption check exploits the index space of the problem observation, thus neglecting other pieces of information inherent to the observation that can lead to discard graphs much more quickly. A more efficient algorithm for subsumption checking is under construction.

Third, the strategy for retrieving a reusable chunk of knowledge, given the current diagnostic problem, also deserves further attention. In fact, no comparison among different strategies was performed. In other words, we were satisfied with the soundness of the adopted strategy and did not look for an ‘optimal’ strategy. Instead, several strategies should be compared so as to provide experimental evidence of their distinct performances and, possibly, find out the best strategy (or the best strategy given a class of problems). In fact, a strategy is more efficient than another if the sum of the time needed for isolating a reusable chunk of knowledge and of the time needed for processing such a chunk, in order to extract the solution of the new problem, is minimal. In particular, several criteria for choosing the 'best' knowledge graph among several graphs of the same type, all matching the current problem, can be proposed, each having its pros and cons. In order to face the matter systematically, such criteria should be implemented and their performances compared by running each of them on the same test suite.

Fourth, the previous point is inherently linked to the maintenance of the knowledge base. The strategy adopted in the implemented environment for finding out a reusable graph, if any, analyzes first maps, then abductions, finally behaviors, where, within the set of the graphs of the same type, spaces are considered before observation-dependent graphs, these graphs being processed in decreasing order with respect to the 'size' of the index space. This order is such that it is useless to save any new chunk of knowledge that shares the same type of the existing graph from which it was derived, since the new chunk will never be used. This discipline for deciding whether or not to save a new chunk of knowledge (the chunk is only saved if its type is different from that of the chunk of knowledge from which it was derived, for instance, if the new graph is a map while the chunk from which it was derived is an abduction) acts as a maintenance action as well as it limits the increase of the size of the knowledge base. If, instead, the strategy analyzed the graphs in a different order (for instance, first maps, then abduction, finally behaviors, where, within each set of graphs of the same type, graphs are in increasing order with respect to the 'size' of the index space), each specific piece of knowledge generated for solving new problems could potentially be exploited in subsequent diagnostic sessions. However, this may result in a huge number of knowledge graphs in the knowledge base and, therefore, in the increase of the time for choosing the chunk of knowledge to be reused in a diagnostic session. This remark points out the need for knowledge maintenance. Although the user category 'knowledge supervisor' is currently in charge of this task, the topic needs further research both for manual and automatic handling.

Finally, the current implementation does not take advantage of the following property (that can be trivially proven): a behavior space can be exploited for solving diagnostic problems having as an initial state not only the system state taken as a starting point by the behavior-space generation algorithm but also any other quiescent state contained in the behavior space itself. In other words, each quiescent state of a behavior space can be regarded as an initial state. Therefore, a behavior space, generated by one construction process, can be seen from the perspective of several initial states. In addition, the behavior space can be used for drawing the abduction and map spaces corresponding to any of its quiescent states. The exploitation of this property will be undertaken in a next version of the diagnostic environment.

11. CONCLUSION

This paper describes both a diagnostic method for performing *a posteriori* diagnosis of CANs and a software environment that embodies it. The current version of the diagnostic environment is a research proof-of-concept prototype that can provide useful feedback for the implementation of an engineered software system exhibiting the same functionalities. The diagnostic method substantiates the latest developments of the so-called active system approach [26] to model-based diagnosis of discrete-event systems, by supporting the compilation of knowledge, both on-line and off-line, and its reuse.

According to other approaches in the literature, compiled knowledge is produced off-line only, once and for all, before any diagnostic problem is considered, then such a knowledge is exploited several times on-line, and it never changes. The current approach, instead, suggests how to increase the (possibly null) compiled knowledge generated beforehand by progressively adding to it the (intermediate and/or final) graph-based data produced on-line when solving diagnostic problems.

The proposed shift of perspective charges the diagnostic process with further responsibilities: exploiting available knowledge (if any) and generating new knowledge. Exploiting available knowledge means being able to ascertain whether a new problem is similar to a previous problem, whose solution has been persistently saved. Note that *similar* does not mean *identical*, it rather means that the solution of an old problem includes the solution of the new problem, in which case the old problem is said to subsume the new. The rationale behind this is that the overhead of this subsumption check, followed by the effort needed for processing the old solution in order to get the new solution, is worthwhile from an efficiency point of view, that is, it results in shorter on-line response times. This theoretical claim has been proved by running the diagnostic environment on meaningful case studies. Experimental results indicate a substantial time gain when solving a compiled problem (by exploiting compiled knowledge) with respect to a crude problem (when no compiled knowledge can be exploited). Moreover, the solution of one problem can be exploited for solving several subsequent subsumed problems. The wider the set of stored solutions, the higher the likelihood of reuse.

Multiple updates and extensions of the current version of the diagnostic environment can be foreseen. First of all, the software system should be aligned with the underlying theoretical research [27], according to which a reusable chunk of compiled knowledge can be obtained by joining proper pieces of knowledge inherent to the subsystems belonging to a generic system partition. The knowledge inherent to a subsystem can, in turn, be generated by joining the pieces of knowledge inherent to its own subsystems, and so on. Thus, knowledge can be generated by following upward a decomposition hierarchy wherein each node corresponds to a piece of knowledge. Such a hierarchy is a graph, that is, any node replication is avoided.

Modularity is, in principle, synergetic with reusability since the decomposition of the knowledge generation process can be performed in such a way that the knowledge required by one or more nodes is already available from previous processing. The cooperation between modularity and reusability is a feature still to be added to the reasoning mechanisms of the developed diagnostic environment.

Moreover, the benefits of a new algorithm for generating the index space in one shot [30], instead of the two-step process mentioned in Section 8.2, should be tested.

Finally, the exploitation of knowledge compilation and reuse for carrying out another diagnostic task, namely monitoring-based diagnosis [31] of CANs, will be the subject of future investigation.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers, whose valuable comments and thorough suggestions helped us to improve the content and presentation of this article.

REFERENCES

1. Struss P. Fundamentals of model-based diagnosis of dynamic systems. *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, Nagoya, Japan, 1977. Morgan Kaufmann: San Francisco, CA, 1997; 480–485.
2. Brusoni V, Console L, Terenziani P, Theseider Dupré D. A spectrum of definitions for temporal model-based diagnosis. *Artificial Intelligence* 1998; **102**(1):39–80.
3. Lamperti G, Pogliano P. Event-based reasoning for short circuit diagnosis in power transmission networks. *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, Nagoya, Japan, 1977. Morgan Kaufmann: San Francisco, CA, 1997; 446–451.
4. Pencolé Y. Decentralized diagnoser approach: Application to telecommunication networks. *Proceedings of the 11th International Workshop on Principles of Diagnosis (DX'00)*, Morelia, Mexico, 2000; 185–192.
5. Rozé L, Cordier MO. Diagnosing discrete-event systems: Extending the 'diagnoser approach' to deal with telecommunication networks. *Journal of Discrete Event Dynamic Systems: Theory and Application* 2002; **12**:43–81.
6. Console L, Picardi C, Ribaudo M. Process algebras for systems diagnosis. *Artificial Intelligence* 2002; **142**(1):19–51.
7. Lamperti G, Zanella M. A bridged diagnostic method for the monitoring of polymorphic discrete-event systems. *IEEE Transactions on Systems, Man, and Cybernetics—Part B: Cybernetics* 2004; **34**(5):2222–2244.
8. Pencolé Y, Cordier MO. A formal framework for the decentralized diagnosis of large scale discrete event systems and its application to telecommunication networks. *Artificial Intelligence* 2005; **164**:121–170.
9. Sampath M, Sengupta R, Lafortune S, Sinnamohideen K, Teneketzis DC. Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control* 1995; **40**(9):1555–1575.
10. Sampath M, Sengupta R, Lafortune S, Sinnamohideen K, Teneketzis DC. Failure diagnosis using discrete-event models. *IEEE Transactions on Control Systems Technology* 1996; **4**(2):105–124.
11. Sampath M, Lafortune S, Teneketzis DC. Active diagnosis of discrete-event systems. *IEEE Transactions on Automatic Control* 1998; **43**(7):908–929.
12. Zad SH, Kwong RH, Wonham WM. Fault diagnosis in timed discrete-event systems. *Proceedings of the 38th IEEE Conference on Decision and Control (CDC'99)*, Phoenix, AZ, 1999. IEEE: Piscataway, NJ, 1999; 1756–1761.
13. Cassandras CG, Lafortune S. *Introduction to Discrete Event Systems* (*The Kluwer International Series in Discrete Event Dynamic Systems*, vol. 11). Kluwer Academic: Boston, MA, 1999.
14. Lunze J. Diagnosis of quantized systems based on a timed discrete-event model. *IEEE Transactions on Systems, Man, and Cybernetics—Part A: Systems and Humans* 2000; **30**(3):322–335.
15. Debouk R, Lafortune S, Teneketzis D. Coordinated decentralized protocols for failure diagnosis of discrete-event systems. *Journal of Discrete Event Dynamic Systems: Theory and Application* 2000; **10**:33–86.
16. Schullerus G, Krebs V. Diagnosis of a class of discrete-event systems based on parameter estimation of a modular algebraic model. *Proceedings of the 12th International Workshop on Principles of Diagnosis (DX'01)*, San Sicario, Italy, 2001; 189–196.
17. McIlraith SA. Explanatory diagnosis: Conjecturing actions to explain observations. *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, Trento, Italy, 1988. Morgan Kaufmann: San Francisco, CA, 1998; 167–177.
18. Baroni P, Lamperti G, Pogliano P, Zanella M. Diagnosis of large active systems. *Artificial Intelligence* 1999; **110**(1):135–183.
19. Barral C, McIlraith S, Son TC. Formulating diagnostic problem solving using an action language with narratives and sensing. *Proceedings of the 7th International Conference on Knowledge Representation and Reasoning (KR'2000)*, Breckenridge, CO, 2000. Morgan Kaufmann: San Francisco, CA, 2000; 311–322.
20. Console L, Picardi C, Ribaudo M. Diagnosis and diagnosability using PEPA. *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI'2000)*, Berlin, 2000. IOS Press: Amsterdam, NL, 2000; 131–135.
21. Grastien A, Cordier MO, Largouët C. Incremental diagnosis of discrete-event systems. *Proceedings of the 16th International Workshop on Principles of Diagnosis (DX'05)*, Monterey, CA, 2005; 119–124.
22. Rozé L. Supervision of telecommunication network: A diagnoser approach. *Proceedings of the 8th International Workshop on Principles of Diagnosis (DX'97)*, Mont St. Michel, France, 1997; 103–111.
23. Kurien J, Nayak PP. Back to the future for consistency-based trajectory tracking. *Proceedings of the 11th International Workshop on Principles of Diagnosis (DX'00)*, Morelia, Mexico, 2000; 92–100.
24. Bochmann GV. Finite state description of communication protocols. *Computer Networks* 1978; **2**:361–372.
25. Brand D, Zafiropulo P. On communicating finite-state machines. *Journal of ACM* 1983; **30**(2):323–342.
26. Lamperti G, Zanella M. *Diagnosis of Active Systems—Principles and Techniques* (*The Kluwer International Series in Engineering and Computer Science*, vol. 741). Kluwer Academic: Dordrecht, 2003.
27. Lamperti G, Zanella M. Flexible diagnosis of discrete-event systems by similarity-based reasoning techniques. *Artificial Intelligence* 2006; **170**(3):232–297.

28. Lamperti G, Zanella M. Diagnosis of discrete-event systems from uncertain temporal observations. *Artificial Intelligence* 2002; **137**(1–2):91–163.
29. Aho A, Sethi R, Ullman JD. *Compilers—Principles, Techniques, and Tools*. Addison-Wesley: Reading, MA, 1986.
30. Lamperti G, Zanella M. On processing temporal observations in model-based reasoning. *Proceedings of the 2nd MONET Workshop on Model-Based Systems (MONET'05)*, Edinburgh, U.K., 2005; 42–47.
31. Lamperti G, Zanella M. Monitoring-based diagnosis of discrete-event systems with uncertain observations. *Proceedings of the 19th International Workshop on Qualitative Reasoning (QR'05)*, Graz, Austria, 2005; 97–102.