To appear in Runtime Verification 2009

A Combined On-line/Off-line Framework for Black-Box Fault Diagnosis*

Stavros Tripakis

University of California, Berkeley, and Verimag Laboratory**

Abstract. We propose a framework for fault diagnosis that relies on a formal specification that links system behavior and faults. This specification is not intended to model system behavior, but only to capture relationships between properties of system behavior (defined separately) and the faults. In this paper we use a simple specification language: assertions written in propositional logic (possible extensions are also discussed). These assertions can be used together with a combined on-line/off-line diagnostic system to provide a symbolic diagnosis, as a propositional formula that represents which faults are known to be present or absent. Our framework guarantees monotonicity (more knowledge about properties implies more knowledge about faults) and allows to explicitly talk about diagnosability, implicit assumptions on behaviors or faults, and consistency of specifications. State-of-the-art diagnosis frameworks, in particular from the automotive domain, can be cast and generalized in our framework.

1 Introduction

Fault diagnosis: The goal of fault diagnosis is to identify faults in a system. Faults are usually viewed as causes that result in system behavior deviating from nominal behavior. Such causes can be of different kinds, including manufacturing errors, system wear-out or "logical" bugs. A distinction is often made between *fault detection*, the "task of determining when a system is experiencing problems" and *fault diagnosis*, the task of "explaining", or "locating the source of a system fault once it is detected" [5]. We will not be concerned with such distinctions in this paper. Instead, we will intentionally view a fault as any abstract (and unknown) system property, the presence of which is worth identifying. Such identification can be then used for recovery (e.g., shutdown and reboot) or repair

* This work is supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0720841 (CSR-CPS)), the U.S. Army Research Office (ARO #W911NF-07-2-0019), the U.S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, Lockheed-Martin, National Instruments, Thales, and Toyota.

** 545Q, DOP Center, Cory Hall, EECS Department, University of California, Berkeley, CA 94720-1772, USA. Email: stavros@eecs.berkeley.edu.

(e.g., replacement of the faulty parts). What exactly a fault is, often depends on the application at hand.

One approach to fault diagnosis, which can be termed "white-box", consists in the following steps. First, build a global model of the behavior of the system, that includes the nominal, non-faulty behavior, as well as the faulty behavior for all types of considered faults. Interactions between non-faulty and faulty behavior must also be modeled. Then, design an observer that monitors a set of observable variables of the system and tries to deduce, from the observations, how the system behaved. In turn, knowledge about system behavior can be used to identify faults that may have occurred, if any, which ones, and so on. Examples of this approach to diagnosis are the discrete-event system diagnosis framework of [15] or the timed-automata diagnosis framework of [18].

This approach is difficult to apply in practice, the main reason being that building such a global model is a daunting task: this is due to the complexity and size of the model. Moreover, as claimed in [1], "because defect behavior is so variable, a fault model always leaves some faults unmodeled".

In this paper we follow an alternative approach, where only a *partial* view on the relation between faults and behavior is assumed. This view attempts to capture the knowledge that engineers gain from experience in observing system behavior and identifying faults (using different means). For instance, one could have reached the conclusion that when a particular fault F is present then the system manifests a particular property P in its behavior. This knowledge can be used for fault diagnosis: F will be excluded as a possible fault when observed behavior does not satisfy P.

We term this approach *black-box* diagnosis, because it uses no explicit model of the system: neither its non-faulty, nor its faulty behavior. We only use *assertions* that attempt to capture relationships between behaviors and faults. For example, the assertion $F \Rightarrow P$ expresses formally the relationship described informally in the example above, namely, that if fault F is present then the system exhibits behavior that satisfies property P.

Fault diagnosis in the automotive domain: This work has been motivated by fault diagnosis problems coming mainly from the automotive domain (e.g., see [16,7]). There, a lot of effort is put in carefully designing *monitors* and *testers*, for *on-line* and *off-line* diagnosis, respectively (also called *on-board* and *workshop* diagnosis, respectively [16]). These components check whether a given property holds and report whenever this fails. For example, a monitor may check that a specific value stays within some lower and upper bounds. What appears to be missing, however, is a methodology and overall framework that *coordinates* these monitors and testers, and *correlates* their results, in order to achieve diagnosis. The main goal of this paper is to contribute some elements that we believe are useful toward such a methodology, framework, and associated techniques and tools.

We should note that some representations that relate monitor/test results and faults do exist in the automotive domain. These include the so-called *D-*

matrix, as well as *diagnostic manuals*, typically used by the workshop mechanics [11,4]. D-matrices are a somewhat formal representation that, although useful in some cases, is much less flexible than the representation we propose in this paper, as discussed in Section 4. Diagnostic manuals, on the other hand, contain information such as "if error code C is on, then check components X, Y and Z [for possible faults]". "Error code C on" means a specific monitor (or test) produced a result indicating a fault may be present. It is however difficult, if not impossible, to interpret diagnostic manuals as providing any type of deterministic relation between faults and monitor/test results. One cannot guarantee that when an error code is on then one of the components to be checked will indeed be faulty (it may be the case that none of them is faulty). One cannot guarantee either that the components that are *not* to be checked are non-faulty. For these reasons, we do not discuss diagnostic manuals in this paper.

Our framework: In a nutshell, our framework contains the following elements:

- First, the user designs a set of *on-line monitors* and *off-line testers*. Each monitor, as well as each tester, is responsible for checking a certain property P_i on the observable behavior of the system. The difference between a monitor and a tester is the following. A monitor is *passive*, in the sense that it does not drive the inputs of the system under observation, but only observes a set of observable outputs. A tester, on the other hand, is *active*: it drives the inputs and at the same time observes the outputs.
  One of the benefits of our framework is that these monitors/testers can be designed using different languages, methods and tools. They could, for instance, be "manually" written in a general-purpose programming language, or in a language specifically designed for testing, such as TTCN [8] or *e* [6]. Alternatively, they could be automatically generated from a formal specification, for instance, a Mealy machine [10], an input-output labeled transition system [17], a temporal logic formula [3], a timed automaton [9], and so on. Our framework is independent from how the monitors/testers are built.
- Second, the user provides a formal description of the relationships between the properties P_i and possible faults in the system. This description is given using the *behavior-fault assertion* language (BFA) that we describe in Section 2.
- Third, our framework provides a *diagnoser* and a *fault knowledge base*. The BFA specification is stored in the fault knowledge base. The diagnoser uses the BFA specification and the outputs coming from the monitors or testers to produce a *diagnosis*. The latter is a formula characterizing the current knowledge about faults in the system, i.e., which faults are present, absent, or unknown.
  Our framework combines on-line and off-line diagnosis. On-line, the diagnoser produces a result which is updated dynamically, as the outputs of the monitors are updated, as the behavior of the system evolves. Off-line, tests can be performed on the system, to get additional information, especially about properties that could not be checked on-line.

The approach is illustrated in Figure 1. M_i denote the on-line monitors and T_i denote the off-line testers.

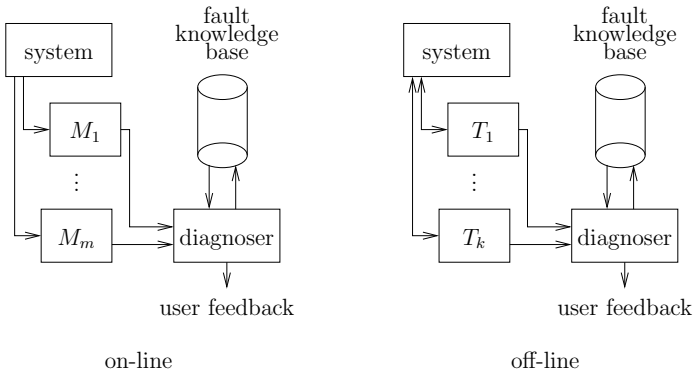

Fig. 1. On-line and off-line diagnosis.

Our framework can seamlessly handle different types of faults, including *permanent* faults (that, once they occur, they stay present) and *transient* or *intermittent* (that may change their presence/absence status as system behavior evolves). This is particularly true in the on-line diagnosis phase, where the output of the diagnoser continuously evolves as the monitor outputs evolve.

2 The Framework

2.1 Behavior-Fault Assertions

We propose a language that allows to explicitly and formally capture relations between observable behavior and faults. This language is essentially propositional logic, where atoms are either propositions F_i, denoting presence of fault i, or propositions P_i, corresponding to properties expressed on the observable behavior. There can be as many F_i and P_i atomic propositions as necessary, but a finite number of each. Notice that each fault proposition F_i or property proposition P_i is simply a symbol (i.e., a propositional variable) at this point. Semantics will be assigned below.

A boolean expression on F_i and P_i is called a *behavior-fault assertion* (BFA).

Examples of BFAs: We will use the symbols $\wedge$, $\vee$, $\neg$, $\Rightarrow$ and $\Leftrightarrow$, to denote logical conjunction, disjunction, negation and implication. $A \Leftrightarrow B$ is shorthand for $A \Rightarrow B \wedge B \Rightarrow A$. Equivalence of logical formulae is denoted $\equiv$. We will use *true* and *false* for the boolean truth values. Finally, we will use $:=$ to define notation.

Here are some examples of BFAs, with their intuitive meaning:

- $F_1 \Rightarrow P_1$: if fault F_1 is present, then the observable behavior must satisfy property P_1.
- $\neg P_1 \Rightarrow \neg F_1$: if the observable behavior falsifies P_1 then fault F_1 is absent (note that this is equivalent to the above).
- $P_1 \vee F_1$: if the observable behavior falsifies P_1 then fault F_1 is present, i.e., equivalent to $\neg P_1 \Rightarrow F_1$.
- $(P_1 \wedge P_2) \Rightarrow (F_1 \vee F_2)$: if the observable behavior satisfies both P_1 and P_2, then at least one of the faults F_1 or F_2 must be present.
- $P_1 \wedge P_2$: this is an assumption on observable behaviors: all observable behaviors must satisfy P_1 and P_2.
- $\neg(F_1 \wedge F_2)$: this is an assumption on faults: if fault F_1 is present then F_2 cannot be present and vice versa.

2.2 Semantics

The system to be diagnosed is typically a dynamical system, that is, it has some (global) state s that evolves with time. Let $\mathcal{S}$ be the set of all possible system states. Let $s(t)$ denote the state of the system at time t. We will not be concerned with what the nature of time is or what the dynamics of the system are (e.g., discrete, continuous, etc.).

Suppose there are n faults that we are interested in. This is reflected by having n propositional symbols corresponding to faults, $F_1, ..., F_n$. At a given system state s, every fault is either *present* or *absent*. We write $s(F_i) = \mathit{true}$ to denote that F_i is present at state s, and $s(F_i) = \mathit{false}$ to denote that F_i is absent at state s. The boolean vector $f = (s(F_1), s(F_2), ..., s(F_n))$ which gives the status of each fault at a given state is called the *fault configuration*. We denote by f_i the i-th element of vector f. As the state evolves with time, so does the fault configuration. We write $f(t)$ to denote the fault configuration at time t. Ideally, we would like to know what $f(t)$ is at any given time t. This is the ultimate goal of diagnosis, albeit not always achievable.

A *behavior* of the system is a history of its state evolution over a period of time. Let $\mathcal{X}$ be the set of all possible system behaviors. We assume that for each property symbol P_i there exists a function $\mathcal{P}_i : \mathcal{X} \rightarrow \{\mathit{true}, \mathit{false}, ?\}$. Function $\mathcal{P}_i$ models the monitor/tester for property P_i. Given a behavior $x \in \mathcal{X}$, $\mathcal{P}_i(x) = \mathit{true}$ means "x satisfies P_i", $\mathcal{P}_i(x) = \mathit{false}$ means "x does not satisfy property P_i", and $\mathcal{P}_i(x) = ?$ means "don't know", that is, x may or may not satisfy P_i.

The ? option is useful to capture the fact that a monitor/tester for P_i may only have *partial observation* capabilities, that is, it may not have access to the entire behavior x, but only part of it. This can be modeled in the semantical function $\mathcal{P}_i$ which can issue ? for x. There are other useful applications of ?, for instance, when properties are expressed in a temporal logic such as LTL [13]. The semantics of LTL are usually defined in terms of infinite behaviors. In monitoring, however, observed behaviors are finite. For this reason, it is not always possible to determine whether an observed behavior satisfies an LTL property or not, and "don't know" needs to be issued as an answer [3].

The three-valued vector $p = (\mathcal{P}_1(x), \mathcal{P}_2(x), ..., \mathcal{P}_n(x))$ is called the *property configuration*. As the behavior of the system evolves with time, so does the property configuration. The property configuration at time t is denoted $p(t)$. We denote by p_i the i-th element of vector p. If p does not contain unknown values, that is, for all $i = 1, ..., n$, $p_i \neq ?$, then p is called *determinate*.

Let Φ be a BFA, f a fault configuration, and p a property configuration. Φ is a propositional logic formula over variables F_i and P_i. Let $\Phi[f, p]$ be the formula obtained by substituting every variable F_i by the truth value f_i, and every variable P_i such that $p_i \neq ?$, by the truth value p_i. For example, let $\Phi_1 = F_1 \Rightarrow P_1 \wedge F_2 \Rightarrow P_2$. Let $f = (\mathit{false}, \mathit{true})$ and $p = (\mathit{false}, ?)$. Then $\Phi_1[f, p] \equiv \mathit{false} \Rightarrow \mathit{false} \wedge \mathit{true} \Rightarrow P_2 \equiv P_2$. Now let $f' = (\mathit{true}, \mathit{true})$. Then $\Phi_1[f', p] \equiv \mathit{true} \Rightarrow \mathit{false} \wedge \mathit{true} \Rightarrow P_2 \equiv \mathit{false}$.

A BFA Φ is *valid* with respect to a system if for any time t, $\Phi[f(t), p(t)]$ is satisfiable.

2.3 The Diagnosis Problem

The *diagnosis*, $\Phi(p)$, is defined to be the set of all fault configurations that are *consistent* with given property configuration p, that is:

$$\Phi(p) = \{f \mid \Phi[f, p] \text{ is satisfiable}\}.$$

The objective of the diagnostic system is to compute, and represent in a compact manner, $\Phi(p(t))$, at any given time t for which a diagnosis is requested by the user of the diagnostic system.

2.4 BFA Consistency

A BFA may be inconsistent, that is, unsatisfiable when interpreted as a classical (two-valued) propositional logic formula. For example, $P_1 \wedge \neg P_1$ is inconsistent.

We require all BFAs to be consistent. This is because we assume that for every property P_i, there exists at least one behavior $x \in \mathcal{X}$ such that $\mathcal{P}_i(x) \neq ?$. This assumption is reasonable: if a property P_i is such that for all $x \in \mathcal{X}$, $\mathcal{P}_i(x) = ?$, then P_i does not serve any purpose since it never gives any useful information. Thus P_i is redundant and can be eliminated from the BFA.

Given the above assumption, since x is a possible behavior of the system, it can be observed, and when it is, the truth value $\mathcal{P}_i(x)$ (i.e., *true* or *false*) will be assigned to P_i. Then, the BFA will evaluate to *false*, which means it is invalid. In other words, with the above assumption, inconsistent BFAs are invalid BFAs, thus should not be considered. Consistency (satisfiability) of a BFA can be checked using a SAT solver.

A BFA can also be inconsistent because of the fault propositions. For example, $F_1 \wedge \neg F_1$ is inconsistent. Clearly, such BFAs should also be discarded, since, by definition of the semantics, every fault is either present or absent, but not both.

2.5 Assumptions Encoded in a BFA

BFAs allow assumptions on either observable system behavior or faults to be explicitly stated. For instance a BFA of the form

$$(P_1 \Rightarrow P_2) \wedge \neg(F_1 \wedge F_2) \wedge \cdots$$

explicitly states an assumption $P_1 \Rightarrow P_2$ on behavior and an assumption $\neg(F_1 \wedge F_2)$ on faults. The former states that if property P_1 holds then property P_2 also holds. This encodes something known about the behavior of the system. Notice that, given observed behavior x such that $\mathcal{P}_1(x) = \mathit{true}$, this assumption allows us to conclude that P_2 also holds, even when $\mathcal{P}_2(x) = ?$. Therefore, using BFA $P_1 \Rightarrow P_2 \wedge P_2 \Rightarrow F_1$, and observing that P_1 holds, allows us to conclude that fault F_1 is present, independently of what the result of the monitor of property P_2 is.[1]

The part $\neg(F_1 \wedge F_2)$ states an assumption on faults, namely, that faults F_1 and F_2 cannot both be present at the same time. This is useful to encode assumptions about the fault model, for instance.

Often assumptions may be "hidden" in a BFA. For example:

$$F_1 \Rightarrow P_1 \wedge F_2 \Rightarrow \neg P_1$$

implies the assumption $\neg(F_1 \wedge F_2)$ on faults and:

$$P_1 \Rightarrow F_1 \wedge P_2 \Rightarrow \neg F_1$$

implies the assumption $\neg(P_1 \wedge P_2)$ on behavior.

Hidden assumptions may be unintentional, so it is important to *extract* them from the BFA and present them to the user. This can be done automatically: the assumptions can be derived from a BFA using existential quantification. Consider a BFA Φ. The hidden assumptions on faults in Φ are characterized by the propositional formula

$$\exists P_1, \exists P_2, ..., \exists P_m : \Phi$$

and the implicit assumptions on behavior by the formula

$$\exists F_1, \exists F_2, ..., \exists F_n : \Phi.$$

Notice that the former is a formula on $F_1, ..., F_n$ variables and the latter is a formula on $P_1, ..., P_m$ variables.

[1] What happens if there is a behavior x for which we observe $\mathcal{P}_1(x) = \mathit{true}$ and $\mathcal{P}_2(x) = \mathit{false}$? This means that the assumption $P_1 \Rightarrow P_2$ made on system behaviors is wrong, therefore, the BFA is incorrect. See discussion on "Assertion validation" in Section 6.

2.6 The Symbolic Diagnoser

A diagnoser is an implementation of the diagnostic system, that is, one particular way of computing and representing in a compact manner the set $\Phi(p(t))$. We provide one possible diagnoser here, but others may exist as well.

Let Φ be a BFA and p be the property configuration at a given time. The diagnoser takes Φ and p as inputs, and produces as output a propositional logic formula $\Psi(\Phi, p)$ over variables F_i. This formula characterizes all fault configurations in $\Phi(p)$ (Theorem 1). $\Psi(\Phi, p)$ is called *the symbolic diagnosis* and it can be computed as follows:

1. For every property P_i such that $p(P_i)$ is not ?, i.e., $p(P_i) \in \{true, false\}$, the truth value $p(P_i)$ is substituted in Φ in the place of symbol P_i. Call the resulting formula $\Phi[p]$. The latter is a formula over variables F_i and possibly also some variables P_i, those for which $p(P_i) = ?$.
2. If $\Phi[p]$ contains no P_i variables, then $\Psi(\Phi, p) := \Phi[p]$. Otherwise, let $P_{i_1}, ..., P_{i_k}$ be the property variables appearing in $\Phi[p]$. Then $\Psi(\Phi, p)$ is obtained by eliminating these variables by existential quantification:

$$\Psi(\Phi, p) := \exists P_{i_1}, ..., \exists P_{i_k} : \Phi[p]$$

The correctness of this diagnoser is stated below.

Theorem 1 (Soundness and completeness). *$f \in \Phi(p)$ iff f satisfies $\Psi(\Phi, p)$.*

For example, let $\Phi_1 = F_1 \Rightarrow P_1 \wedge F_2 \Rightarrow P_2$ and $p = (false, true)$. Then, using the above procedure, we compute:

$$\Psi(\Phi_1, p) \equiv \Phi_1[p] \equiv F_1 \Rightarrow false \wedge F_2 \Rightarrow true \equiv \neg F_1.$$

Indeed, from the fact that property P_1 is *false* and the implication $F_1 \Rightarrow P_1$, we can deduce that fault F_1 is absent.

Next, let $p' = (false, ?)$. Using the above procedure, we compute:

$$\Psi(\Phi_1, p') \equiv \exists P_2 : \Phi_1[p'] \equiv \exists P_2 : F_1 \Rightarrow false \wedge F_2 \Rightarrow P_2 \equiv \neg F_1.$$

Thus, again we can conclude that fault F_1 is absent.

2.7 Ideal Diagnosability and Weaker Notions

Ideally, we would like to identify exactly which faults are present and which are not. This means that ideally $\Phi(p)$ should be a singleton. Obviously, this depends on the observed behavior, captured by the property configuration p. If the observed behavior is completely *uncontrollable* (e.g., produced by a "passive" on-board monitoring framework, passive in the sense that it does not provide inputs to the system) then there is not much we can do. For example, in the BFA Φ_1 of the example above, if for all $x \in \mathcal{X}$ we have $\mathcal{P}_2(x) = true$, then Φ_1 cannot give us information about the status of fault F_2. Moreover, observations

are sometimes inconclusive, that is, for a given $x \in \mathcal{X}$ and property P_i, we have $\mathcal{P}_i(x) = ?$. In this case, no information on P_i is available.

On the other hand, sometimes behaviors are *controllable*, in the sense that we can *subject* the system to predefined tests that are guaranteed to provide an answer to whether a property holds or not. These tests are designed to drive the inputs of the system and observe how the system responds in a given scenario. This is the case, for instance, in *off-line* (also called *workshop*) diagnosis [16].

Inspired by this, we introduce a notion of *ideal diagnosability*, that attempts to capture whether a given BFA can *in principle* provide precise information about which faults are present and which are absent, provided the truth value of every property P_i is known.

Formally, we say that a BFA Φ *allows ideal diagnosability* if for any determinate property configuration p, $\Phi(p)$ is a singleton. Recall that p is determinate if it does not contain unknown values, that is, for all $i = 1, ..., n$ (where n is the length of vector p), $p_i \neq ?$.

For example, let $\Phi_1 = F_1 \Rightarrow P_1 \wedge F_2 \Rightarrow P_2$. Then Φ_1 does not allow ideal diagnosability, because $\Phi_1((true, true))$ obviously contains more than one fault configuration (in fact it contains all of them). On the other hand, the BFA $\Phi_2 = F_1 \Leftrightarrow P_1 \wedge F_2 \Leftrightarrow P_2$, allows ideal diagnosability. We can see that ideal diagnosability requires a pretty complete specification of the relation between properties and faults, which obviously is not always available. For this reason, we do not require BFAs to allow ideal diagnosability in general.

A straightforward, albeit inefficient, way to check whether a given BFA Φ allows ideal diagnosability is to enumerate all possible determinate property configurations (there is 2^m of them, assuming there are m properties) and check, for each such configuration p, whether $\Psi(\Phi, p)$ has a unique solution. Checking whether a propositional logic formula has a unique solution can be done by running a SAT solver twice on the formula, the second time adding the negation of the solution found in the first run, if any.

Weaker notions of diagnosability could also be defined. For example, one may wish to know whether a given BFA Φ allows to determine, in principle, whether at least one of two faults F_1 and F_2 is present. This means that, for any determinate property configuration p, $\Psi(\Phi, p)$ should imply either $F_1 \vee F_2$ or $\neg(F_1 \vee F_2)$.

2.8 Monotonicity

One nice property of our framework is that it is *monotonic*, in the sense that more knowledge about properties implies more knowledge about faults. Let us make this precise.

Consider two property configuration vectors $p = (p_1, ..., p_n)$ and $p' = (p'_1, ..., p'_n)$. We say that p *is more determinate than* p', noted $p \leq p'$, if for all $i = 1, ..., n$, $p_i =?$ implies $p'_i =?$. Intuitively, $p \leq p'$ means that p provides more knowledge than p'.

Theorem 2 (Monotonicity). *$p \leq p'$ implies $\Phi(p) \subseteq \Phi(p')$.*

3 On-line and Off-line Diagnosis

As illustrated in Figure 1, we envisage a diagnostic system which combines *on-line* diagnosis (also called *on-board* diagnosis in the automotive domain) with *off-line* diagnosis (also called *workshop* diagnosis in the automotive domain [16]).

3.1 On-line monitoring and diagnosis

The on-line diagnostic system consists of several components:

First, a set of monitors, denoted $M_1, ..., M_m$ in Figure 1, where M_i monitors property P_i. At any given point in time t, M_i outputs $\mathcal{P}_i(x(t))$, where $x(t)$ is the behavior of the system up to time t. Together the outputs from all monitors form the current property configuration vector $p(t)$.

Second, a *fault knowledge base*. The latter contains a BFA Φ, which is an input to the diagnostic system, designed by the user, and the current symbolic diagnosis, $\Psi(\Phi, p(t))$, which is updated dynamically by the diagnoser.

Third, the on-line diagnosis system includes a diagnoser. The diagnoser takes as input the BFA Φ and the current property configuration $p(t)$, computes the current diagnosis, $\Psi(\Phi, p(t))$, and stores the latter in the fault knowledge base. It may also provide feedback to the user, e.g., telling the user that it is necessary to visit a workshop for further, off-line diagnosis. How this feedback is generated is beyond the scope of this paper. How often the current property configuration and current diagnosis are updated depends on the particular needs of the system, which in turn depend on the application.

3.2 Off-line testing and diagnosis

The off-line diagnostic system is not very different from the on-line one. The only difference is that monitors are replaced by testers. A tester T_j is supposed to test a certain property $P_{i(j)}$. There may be more than one testers for the same property, and no testers for some properties. Typically, during off-line diagnosis, one would like to test properties for which no conclusion was reached during the on-line phase, that is, properties P_i such that $\mathcal{P}_i(x) = ?$. On the other hand, there may be properties for which on-line monitoring is guaranteed to be conclusive, and these may require no off-line tests.

The off-line tests are executed, either in parallel, or in sequence, or using a mix of both strategies. How exactly this is done depends on the application, and in particular on the type of testing strategy that the testing harness permits. An interesting problem here is *test sequencing*, discussed below.

One assumption we make is that during execution of the tests the fault configuration of the system does not change. This is important, since, contrary to on-line monitoring, the tests are performed only for a given amount of time and not continuously. If faults change during test execution, then the result of a test does not mean much, since it could be different if a test were to be executed again.

Once the tests are run, a new property configuration vector is obtained. This is used, as in on-line diagnosis, to update the symbolic diagnosis.

Test sequencing: Off-line diagnosis raises the problem of *test sequencing*, namely, what is the "best" order in which tests should be run. It is crucial to define carefully what exactly "best" means, and many different variants of this problem have appeared in the literature (e.g., see [12]). Here, we define and study a simple variant that fits our logical framework.

Executing tests takes time and other resources, i.e., it is a costly process. Therefore, there is an incentive to minimize the number of tests that need to be run. In the ideal diagnosability case, running all the tests is guaranteed to produce a definite answer as to the status of each and every fault. For example, consider the simple case where there is a single fault F_1, and the BFA $F_1 \Leftrightarrow (P_1 \Rightarrow P_2 \wedge \neg P_1 \Rightarrow P_3)$. This BFA clearly allows ideal diagnosability. Suppose there is one test for each property, i.e., test T_i for P_i, for $i = 1, 2, 3$. In which order should the tests be run?

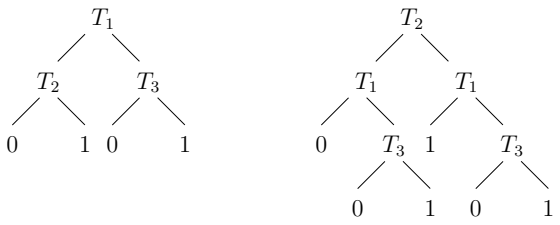

Fig. 2. Test execution strategies as decision trees.

Two possible execution strategies are shown in Figure 2. Each strategy is in essence a *decision tree*: node T_i of the tree corresponds to executing test T_i; the left (respectively, right) branch is followed when T_i produces a value indicating that P_i is *false* (respectively, *true*); a leaf of the tree labeled 0 means that fault F_1 is absent, and 1 means that F_1 is present.

Intuitively, the strategy represented by the left-most tree is better: the tree is smaller, and it is guaranteed to require execution of at most two tests in the worst case; on the other hand, the right-most tree may require execution of three tests, thus represents a generally more costly strategy. A reasonable definition of "best" test strategies could therefore be *"representable by a decision tree of minimal depth"*. Unfortunately, finding a strategy representable by a tree of minimal depth is an intractable problem. Indeed, satisfiability of a propositional formula ϕ can be easily reduced to the problem of checking whether there exists a tree of depth 1 representing ϕ: such a tree exists iff ϕ is unsatisfiable or valid.

Similar test sequencing problems could be defined for weaker notions of diagnosability.

4 Application: Capturing the D-matrix

One of the representations often employed in fault diagnosis contexts, in the automotive but also other domains, is the so-called *D-matrix* (sometimes called the "diagnostic matrix" [11] and sometimes the "dependency matrix" [4]). In this section, we compare D-matrices to BFAs. We show that capturing D-matrices as BFAs is straightforward, while this is not true in the opposite direction.

The rows of a D-matrix correspond to faults (or "components" that may be faulty or non-faulty) and the columns correspond to *tests*. In the boolean version of the D-matrix, an entry (i, j) of the matrix contains the expected outcome of test j, *given* that fault i is present. For instance, if test j has a boolean outcome, then (i, j) is either 0 or 1. In some cases the expected outcome of a test is uncertain (that is, it may be either 0 or 1, but we do not know in advance). In these cases the entry (i, j) may contain a special "?" value.

It is important to note that usually the D-matrix representation makes a single-fault assumption, in the sense that only one of the faults is assumed to be present. This allows to obtain the values of the entries of the matrix, for example, by "injecting" a fault, executing the tests, and observing their outcome. Only a single fault is injected at any time. The matrix may also contain a special row corresponding to "no fault", containing the test outcomes when no fault has been injected.

The single-fault assumption is also used when checking diagnosability. The latter holds iff there are no two rows in the matrix such that their entries "match" for every column. Two entries match if they have identical value or if one of the values is ? (because this means the outcome can be anything). If rows i_1 and i_2 match then it is possible (for some test outcomes, that we cannot a-priori predict) that faults i_1 and i_2 cannot be distinguished. On the other hand, if no rows match, then by executing all the tests we should be able to identify the (single, by assumption) fault whose row does not contradict the test outcomes. Note that it is possible that all rows contradict the fault outcomes. This implies either that the D-matrix is incorrect, or that there are additional faults (rows) that have not been included in the matrix.

There are also probabilistic versions of the D-matrix representation. In the probabilistic version, the entry (i, j) is a conditional probability distribution over the test of possible outcomes of test j. For example, if there are two outcomes 0 and 1, the entry (i, j) could be $(0.1, 0.9)$ for a 90% probability to get an output 1 when executing test j and 10% probability to get 0, given that the fault is i. We will not consider probabilistic D-matrices in this paper.

A boolean D-matrix M can be easily captured in terms of a BFA: an entry $M(i, j) = b$, with $b \neq ?$, can be represented as the assertion $F_i \Rightarrow P_j$, where P_j is the property associated with test j. $\mathcal{P}_j$ is defined as follows. Given a behavior x, if the inputs of x do not match the inputs that test j would provide, then $\mathcal{P}_j(x) = ?$. Otherwise: if the outcome of test j on x is b, then $\mathcal{P}_j(x) = \mathit{true}$, otherwise $\mathcal{P}_j(x) = \mathit{false}$. Practically speaking, the tester for P_j implements exactly the test j, and if a monitor is needed instead, it can be derived from an implementation of test j by turning outputs of the test into monitored inputs.

On the other hand, capturing BFA assertions in a D-matrix representation can be non-straightforward, as well as expensive. For example, in order to capture the assertion $F_1 \Rightarrow (P_1 \vee P_2)$ one would have to devise a test that checks whether the disjunction of properties P_1, P_2 holds and add a corresponding column to the matrix (notice that almost all entries of the column will be ?, except for the entry corresponding to F_1, which is wasteful). If there are already tests covering properties P_1 and P_2 individually (presumably because these are used in other assertions) then: (1) one would need to ensure consistency of the information stored in the matrix (for each row, the value of the entry of $P_1 \vee P_2$ must not contradict the disjunction of the values of P_1 and P_2); and (2) many redundant entries are introduced in the matrix.

As another example, in order to capture the assertion $P_1 \Rightarrow F_1$, one would have to first transform it into the equivalent form $\neg F_1 \Rightarrow \neg P_1$. Then, one would have to add a row in the matrix corresponding to $\neg F_1$, and a column corresponding to a test that checks $\neg P_1$. Again, one would have to maintain consistency between rows F_1 and $\neg F_1$. This can become difficult for more complex assertions involving multiple faults, e.g., $P_1 \Rightarrow (F_1 \vee F_2)$ or $(F_1 \wedge F_2) \Rightarrow P_2$, and so on.

These simple examples show that the BFA model is more flexible and more convenient than the D-matrix model.

5 Related work

Our framework can be seen as a specialization of Reiter's *consistency based diagnosis* framework [14]. The latter is a triple (SD, C, O), where SD is a *system description*, C a set of *components*, and O a set of *observations*. In Reiter's framework, SD and O are general first-order logic formulas, whereas C is a set of constants. We can cast our framework in Reiter's terms as follows: SD will be the BFA Φ, C will be the set of fault propositions $\{F_1, ..., F_n\}$, and O will represent the known values in the property configuration p. For instance, if there are three properties P_1, P_2, P_3 and $p = (true, ?, false)$ then O is $P_1 \wedge \neg P_3$.

Although more general than ours, Reiter's framework has been intended mostly as a "white-box" framework, where the entire structure and behavior of the system is modeled in SD. This is evident both from the terminology used ("system description", "components") as well as from the examples in the original and subsequent papers by Reiter and other authors. By restricting the framework, and treating faults as "first-class objects", we can focus only on the relationships between behavior and faults, and not on an extensive modeling of behavior, which is infeasible in practice. In that sense, our framework can also be seen as an *expert system*, in that it attempts to capture, through BFAs, the knowledge of the engineers.

Another benefit of restricting Reiter's framework is that computing the diagnosis becomes much simpler: our method only uses existential quantification of propositional formulas, and does not rely on conflict sets and hitting sets as in Reiter's method. Another difference is that our framework guarantees monotonicity.

Our framework is also closely related to Bauer's [2]. He also uses separately defined monitors to "source out" behavior and reduce Reiter's first-order logic framework to propositional logic. Causal and structural information still remain in *SD*, however. We go one step further and make no assumptions on how the relationships between behavior and faults are specified. In [2], diagnoses are computed by LSAT, a specially-constructed SAT-solver that can produce multiple solutions given a user-provided n-fault assumption. In our framework, diagnoses are represented symbolically as a propositional formula on the fault variables F_i.

6 Conclusions and Perspectives

We proposed a fault diagnosis framework that assumes no explicit model of system behavior, but instead relies on a formal specification of the relationships between behavior and "faults" (or fault "causes", or any other notion that may be relevant to a particular application). By decoupling behaviors and faults, we can treat the problem of how to construct monitors for behaviors separately from the problem of how to use monitor outputs to gain knowledge about faults.

Future directions include:

Richer assertion languages: In this paper we considered a simple instance of this framework where the relationships between behaviors and faults are captured in propositional logic. This simple language can be extended in multiple ways. For instance, one possibility is to use *temporal logic*, or some other formalism that allows to talk about ordering or timing of observations. For example, we may want to express that if property P_1 is observed *before* P_2 is observed, then this implies fault F_1 is present, but if P_2 has been observed before P_1 then nothing can be inferred. Again, P_1 and P_2 can be complex (and themselves dynamical) properties, but the top-level specification treats them as atoms.

Probabilistic and statistical learning frameworks: Another possibility is to move from a boolean to a probabilistic interpretation. For instance, we may want to state something of the form: "if fault F_1 is present then there is a 90% probability that test T_1 produces outcome *true* and 10% probability that it produces *false*". Or: "if property P_1 is observed then there is a 50% probability that F_1 is present". Combined with a language that allows us to speak about time, this could become: "if property P_1 has been continuously *true* for at least t time units, then there is a $h(t)$% probability that fault F_1 is present", where $h(\cdot)$ is some function. Probabilistic logics and statistical learning methods could be potentially useful in extending the framework in this direction.

Assertion validation: A BFA is an assertion relating the observed behaviors and the faults. This assertion is constructed presumably by humans, based on their experience or other means, therefore, it may be invalid, that is, erroneous in a variety of different ways. To list a few:

(1) The assumptions hidden in the BFA about system behavior (e.g., $P_1 \Rightarrow P_2$) may be incorrect. This can be discovered when a behavior x is observed, for instance, such that $\mathcal{P}_1(x) = \mathit{true}$ and $\mathcal{P}_2(x) = \mathit{false}$.

(2) Similarly, the hidden assumptions about faults (e.g., $\neg(F_1 \wedge F_2)$) may be incorrect. This can be discovered when the diagnosis obtained by some other means contradicts these assumptions (for example, we find that both faults F_1 and F_2 were present).

(3) Finally, the relationships between behavior and faults are incorrect. This can be discovered when the diagnosis obtained by the BFA and some property configuration contradicts the diagnosis obtained by some other means. For instance, if $\Psi(\Phi, p)$ is $\neg F_2$ but somehow F_2 turns out to be present, then this implies the BFA is invalid.

In all above cases, once invalidity is detected, the problem is how to locate which "parts" of the assertion are incorrect and to correct them. This is of course a difficult problem, of locating errors in a formal model, and it is related to automated debugging.

Acknowledgments

Thanks to Mark Wilcutts and Hakan Yazarel from Toyota, and Ken McMillan and Anubhav Gupta from Cadence, for useful discussions. Thanks to Andreas Bauer from TU Munich for pointing out his work during a discussion at the Dagstuhl 2007 Seminar on Runtime Verification (the ideas outlined in this paper arose independently).

References

1. R.C. Aitken. Modeling the unmodelable: Algorithmic fault diagnosis. *IEEE Design & Test of Computers*, 14(3):98–103, 1997.
2. A. Bauer. Simplifying diagnosis using LSAT: a propositional approach to reasoning from first principles. In *Proc. 2005 Intl. Conf. on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, volume 3524 of *LNCS*, pages 49–63. Springer, June 2005.
3. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*, February 2009.
4. A. Beygelzimer, M. Brodie, S. Ma, and I. Rish. Test-based diagnosis: Tree and matrix representations. In *IM 2005 - IFIP/IEEE International Symposium on Integrated Network Management*, pages 529–542, 2005.
5. M. Chen, A.X. Zheng, J. Lloyd, M.I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Autonomic Computing*, 2004.
6. S. Iman and S. Joshi. *The e-Hardware Verification Language*. Springer, 2004.
7. R. Isermann. Model-based fault detection and diagnosis: status and applications. *Annual Reviews in Control*, 29:71–85, 2005.
8. ISO/IEC. Open Systems Interconnection Conformance Testing Methodology and Framework – Part 1: General Concept – Part 2 : Abstract Test Suite Specification – Part 3: The Tree and Tabular Combined Notation (TTCN). Technical Report 9646, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, 1992.

9. M. Krichen and S. Tripakis. Conformance Testing for Real-Time Systems. *Formal Methods in System Design*, 34(3):238–304, June 2009.
10. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. *Proceedings of the IEEE*, 84:1090–1126, 1996.
11. J. Luo, K. Pattipati, L. Qiao, and S. Chigusa. Towards an integrated diagnostic development process for automotive systems. In *IEEE Intl. Conf. Systems, Man and Cybernetics*, pages 2985– 2990, 2005.
12. K. Pattipati and M. Alexandridis. Application of heuristic search and information theory to sequential fault diagnosis. *IEEE Trans. Systems, Man and Cybernetics*, 20(4):872–887, August 1990.
13. A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
14. R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.
15. M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete event systems. *IEEE Transactions on Automatic Control*, 40(9), September 1995.
16. P. Struss and C. Price. Model-based systems in the automotive industry. *AI Magazine*, 24(4):17–34, 2004.
17. J. Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99*, volume 1664 of *LNCS*. Springer, 1999.
18. S. Tripakis. Fault Diagnosis for Timed Automata. In W. Damm and E.-R. Olderog, editors, *Formal Techniques in Real Time and Fault Tolerant Systems, 7th Intl. Symposium (FTRTFT'02)*, volume 2469 of *LNCS*, pages 205–224. Springer, September 2002.