Appl Intell (2012) 36:511–528
DOI 10.1007/s10489-008-0143-x

Diagnosis and repair of dependent failures in the control system of a mobile autonomous robot

Jörg Weber · Franz Wotawa

Published online: 20 September 2008

Abstract Detecting, locating and repairing faults is a hard task. This holds especially in cases where dependent failures occur in practice. In this paper we present a methodology which is capable of handling dependent failures. For this purpose we extend the model-based diagnosis approach by explicitely representing knowledge about such dependencies which are stored in a failure dependency graph. Beside the theoretical foundations we present algorithms for computing diagnoses and repair actions that are based on these extensions. Moreover, we introduce a case study which makes use of a larger control program of an autonomous and mobile robot. The case study shows that the proposed approach can be effectively used in practice.

Keywords Model-based diagnosis · Repair · Dependent failures · Monitoring

1 Introduction

Mobile autonomous robots have gained increasing importance in the last few years. Scientific progress and the availability of powerful hardware at reasonable costs are contributing to the development of autonomous systems for a wide range of applications including space missions, household tasks like vacuum-cleaning or grass mowing, the care for elderly or even industrial applications (e.g., in the area of logistics).

It is widely recognized that intelligent mobile systems require sensing, acting and planning capabilities in order to perform their tasks in unknown and uncertain environments. However, past experience has shown that a main obstacle towards real autonomy of mobile robots is the occurrence of unforeseen faults during a mission. Failures in the robot's hardware may occur due to unexpected interactions with a harsh environment, whereas failures in the control system are caused by bugs in the software. In practice, the latter cannot be totally avoided, as control systems are often very large and complex and thus not amenable to exhaustive testing or formal verification. Even in the case where the system can be proven to be functionally correct, errors at runtime like out of bounds or out of memory exceptions can occur due to the finite nature of datatypes and resources like memory. For example, someone might prove that a recursive function is correct using induction. However, in a real system the number of recursive function calls is limited by the maximum size of the stack used to store the arguments of each call. Hence, for larger instances the recursive function might fail due to the restrictions of the underlying runtime environment. Unfortunately such faults are usually not detected during programming and testing.

In case of a runtime failure, an intelligent system which is augmented with autonomous runtime diagnosis and repair/reconfiguration abilities has a much better chance to survive and to achieve the desired goals. Past researchers have addressed these problems, but most of these studies have focused on hardware aspects, in particular the diagnosis of sensors and actuators. By contrast, we have tackled the runtime diagnosis of failures in the control software (SW) of mobile autonomous robots [1–3]. We assume that

This research has been funded in part by the Austrian Science Fund (FWF) under grant P20199-N15.

J. Weber (✉) · F. Wotawa
Technische Universität Graz, Institute for Software Technology, Inffeldgasse 16b/2, 8010 Graz, Austria
e-mail: jweber@ist.tugraz.at

F. Wotawa
e-mail: wotawa@ist.tugraz.at

the SW system is decomposed into independent components (modules) which run concurrently and communicate with each other in an asynchronous manner. Runtime faults are detected by continuously monitoring properties of the system. Moreover, we presented a technique which, relying on logical dependencies between properties, is able to locate the failed components. It is crucial that only the localization of a fault allows the diagnosis system to perform a meaningful repair or reconfiguration actions with the aim of fully recovering the capabilities of the control system.

Our runtime diagnosis approach mainly aims at severe faults like crashes or deadlocks. As proposed in [1], our runtime diagnosis system attempts to recover the control system by restarting those components which are assumed to have failed. Although this is not a real repair action, as the same fault may occur again, a practice has shown that failed components often work correctly after a restart. However, sometimes it is not possible to autonomously perform a complete repair. Hence, in [4] we presented an AI-planning system for a mobile autonomous robot which is able to infer the capabilities that are still provided by the remaining software system and which adapts its decision-making according to the degraded functionality.

It should be noted that our work does not deal with failures related to the robot's hardware, i.e., we assume that the hardware is working correctly. Moreover, we do not address sensing failures, failures due to a bad world model of the robot, or failures caused by unexpected physical interactions with the environment.

Our fault localization method uses model-based diagnosis techniques; in particular, we rely on the consistency-based diagnosis paradigm [5–7]. The basic idea of model-based diagnosis is to employ a formalized system description, which is composed of the behavior models of the constituent components, and a set of observations obtained at runtime in order to compute the *minimal diagnoses*. Intuitively, a minimal diagnosis is a subset-minimal set of components which may have failed. In our application, the system description contains abstract behavior models capturing the logical dependencies between system properties.

As usual in model-based diagnosis, the fault localization techniques we presented in [3] assume that components fail independently, i.e., the failure of one component cannot cause the failure of other components. Unfortunately, although this simplifying assumption is reasonable in many applications, there are also domains in which it may lead to unsatisfying results. In particular, we have experienced that dependent failures are quite common in robot control systems. The main reason is the fact that the components of robot control systems are often tightly coupled and thus not entirely independent from each other. For example, components may communicate over channels which do not completely decouple the sender and receiver, e.g., remote procedure calls. Another example is the fact that software components are often based on the same underlying framework (e.g., CORBA), and a fault in this framework may propagate to the dependent components.

One consequence of dependent failures is that the focus on minimal diagnoses is no longer justified, as they often do not contain all components which have failed. If this happens, a repair which is based solely on minimal diagnoses cannot fully recover the system. Moreover, the knowledge of the causal order of multiple dependent failures, which is not reflected in minimal diagnoses, is often crucial for the successful repair of the system.

Therefore, we propose the concept of *diagnosis environments* (*DEs*). The DEs of a diagnosis Δ consider all components as failed which are also in Δ, and, in addition, they may contain further components which may have caused the failure of components in Δ. Furthermore, a DE captures the causal order in which the components have failed. The definition of DEs is based upon the notion of diagnosis, and the computation of the minimal DEs is done after the generation of the minimal diagnoses. The computation of the DEs relies on a *failure dependency graph* which indicates possible paths of failure propagation. This graph is part of the system model and provided by the modeller. Moreover, in order to rule out implausible DEs we augment the system model with fault models capturing the faulty behavior of components which have failed dependently.

We introduced the concept of diagnosis environments in [8], which, to the best of our knowledge, was the first work within the framework of consistency-based diagnosis which proposes a general approach to the issue of dependent failures at the level of logical reasoning. In [9] we provided a formalization of our approach and we presented some theoretical results. Moreover, in [10] we presented an improved formalization and new algorithms.

This paper is organized as follows. The following section gives a brief introduction to the principles of model-based diagnosis. Section 3 presents our approach to the runtime diagnosis in a robot control system. In the following two sections we provide a discussion of dependent failures, and show that such failures are common in robot control systems, but also in other domains. In Sect. 6 we propose the concept of diagnosis environments (DEs), and we provide an algorithm for computing all minimal DEs of a system in Sect. 7. Moreover, Sect. 8 addresses the issue of repair in systems with dependent failures. Section 9 presents several case studies in which our diagnosis system is able to automatically repair a control system after a series of dependent failures. Finally, we discuss related research and provide a conclusion.

2 Background: model-based diagnosis

We follow the model-based diagnosis paradigm, in particular the consistency-based diagnosis approach [5–7]. A collection of seminal papers on model-based diagnosis can be found in [11].

Definition 1 (System) A *system* is a tuple (*SD*, *COMP*, *OBS*). As usual, *SD* is the *system description*, i.e., a set of logical sentences capturing the system behavior. *COMP* is the set of components, and *OBS* is a set of logical sentences representing the observations.

The system description *SD* is typically composed of the behavior models of the single components. In our application, *OBS* comprises observations from the running system.

We illustrate the principles of model-based diagnosis by means of a simple example from everyday life. It is similar to the examples used in [12] and [13]. Consider the circuit in Fig. 1 which consists of a voltage source V and two bulbs, B_1 and B_2 (hence, $COMP = \{V, B_1, B_2\}$). The voltage provided by V is denoted by u_v, and u_{b1} and u_{b2} denote the voltage at the two bulbs. In our abstract model we use only two different qualitative values for the magnitude of a voltage, namely *zero* ($= 0$) or *pos* (> 0).

SD contains the behavior models of the components, i.e., it describes the correct (expected) behavior. For this purpose we use the well-known literal $\neg ab(c)$, meaning that component c is "not abnormal":

$$\begin{aligned} &\neg ab(V) \rightarrow (u_v = pos) \\ &\neg ab(B_1) \rightarrow (fil_1 = intact) \qquad [fil..\text{"filament"}] \\ &(fil_1 = intact) \wedge (u_{b1} = pos) \rightarrow (light_1 = on) \end{aligned} \tag{1}$$

The model of B_2 is analogous to B_1. Moreover, we need to describe in *SD* how the components are connected:

$$\begin{aligned} u_{b1} &= u_v \\ u_{b2} &= u_v \end{aligned} \tag{2}$$

Note that we assume that all wires behave correctly. This corresponds to the *no-faults-in-structure* assumption which is very common in model-based diagnosis approaches [14]. This is not really a restriction of the approach since wires can be represented as components which might also fail.

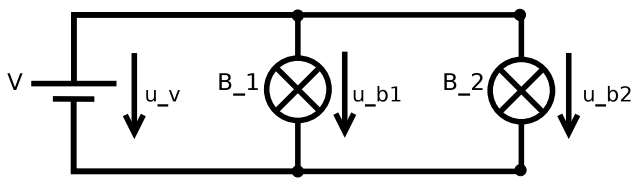


Fig. 1 Sample circuit consisting of a voltage source and two bulbs

Unfortunately such a model leads to additional diagnoses which are seldomly of help for the user in practical applications. Claudia Böttcher [15] presented a solution to this problem by considering problems that occur in practice like the unexpected functionality of an electronical circuit when two wires interact in an unwanted way.

Finally, we also assume that there are domain closure axioms stating that a voltage is either *zero* or *pos*, that a filament is either *intact* or *broken*, and that a light is either *on* or *off*.

The following definition of a *diagnosis* is very similar to the definition given in [5]:

Definition 2 (Diagnosis) A *diagnosis* Δ is a set of components ($\Delta \subseteq COMP$) s.t.

$$SD \cup OBS \cup \{\neg ab(c) \mid c \in COMP \setminus \Delta\} \cup \{ab(c) \mid c \in \Delta\} \tag{3}$$

is consistent. Furthermore, a diagnosis Δ is a *minimal diagnosis* iff no proper subset of it is a diagnosis.

There are well-known algorithms, like Reiter's Hitting Set algorithm [5], which compute all minimal diagnoses for a given system.

For example, suppose we obtain the observations $OBS = \{light_1 = on,\ light_2 = on\}$. Then $SD \cup OBS \cup \{\neg ab(V), \neg ab(B_1), \neg ab(B_2)\}$ is consistent. Hence, there is only one minimal diagnosis: the empty set {}. In other words, it can be reasonably assumed that all components are correct. Note that all power sets of *COMP* (i.e., all elements in 2^{COMP}) are diagnoses, but only the empty set is a minimal one.

Now suppose $OBS = \{light_1 = off,\ light_2 = off\}$. Then there are two minimal diagnoses: $\Delta_1 = \{V\}$ and $\Delta_2 = \{B_1, B_2\}$. I.e., the most plausible explanations of the symptoms are that either the voltage source or both of the bulbs have failed. We can refine the diagnosis by looking whether or not the filaments of the bulbs are broken. E.g., if we gather the additional observations $OBS = \{\ldots, fil_1 = broken, fil_2 = broken\}$, then there is only one minimal diagnosis $\Delta_2 = \{B_1, B_2\}$.

Finally, suppose $OBS = \{light_1 = on, light_2 = off\}$. This yields the minimal diagnoses $\Delta_1 = \{V\}$ and $\Delta_2 = \{B_2\}$. Clearly, this does not correspond to our intuition that a failed voltage source does not produce any voltage. Hence, Δ_1 is obviously implausible, as $light_1 = on$ is observed which is possible only if there is a voltage at B_1. The reason why Δ_1 is not logically refuted is the fact that the model above only specifies a correct behavior and does not make any commitment regarding the behavior in case of a failure. As a consequence, we augment our model by adding a *physical impossibility axiom* [13] stating that a bulb can never produce light when there is no current flowing through it:

$$(light_1 = on) \wedge (u_{b1} = zero) \rightarrow \bot \tag{4}$$

Now we can conclude that u_{b1} must be *pos* and, as a consequence, also u_{b2} is *pos*. Therefore, there remains only one minimal diagnosis, as desired: $\Delta_2 = \{B_2\}$. An alternative way of refuting the implausible minimal diagnosis Δ_1 would be to define an abnormal behavior of components as well, i.e., using the fault mode $ab(c)$ [12]:

$$\begin{aligned} &ab(V) \rightarrow (u_v = zero) \\ &\neg ab(B_1) \wedge (light_1 = on) \rightarrow (u_{b1} = pos) \end{aligned} \tag{5}$$

Again, only Δ_2 remains as minimal diagnosis. In general, physical impossibilities can often be used instead of fault models at the advantage of not increasing computational complexity. The disadvantage of physical impossiblities is that they very likely harm the compositional modeling principle which says that the overall system model should be composed from the component models and the known connections only.

One of the main advantages of model-based diagnosis is the availability of generic algorithms which can be applied to various kinds of systems, provided that a logical model of the system has been created. Another advantage is the fact that model-based diagnosis offers a coherent and well-understood formal framework. Moreover, model-based diagnosis is, unlike many rule-based expert systems, able to deal with multiple faults. Finally, the compositional modelling approach facilitates the creation and maintenance of complex models of very large systems.

Notice that Reiter's framework in [5] only considers the definition of a nominal behavior. The authors of [7] extend these concept to system descriptions which also specify an abnormal behavior.

3 Runtime diagnosis in a robot control system

The authors of [1–3] introduced techniques which enable the model-based fault detection and localization in the control system of a mobile autonomous robot and the repair of a partly failed system. Figure 2 depicts a fragment of the control system of a soccer robot. In addition, it also contains a hardware component, *Ca*.

This architectural view comprises largely independent components and the connections between them. The connections represent data flows over different communication channels. The software components communicate either by exchanging asynchronous events, i.e., chunks of data, over an event channel (see below) or by performing remote procedure calls, i.e., a component c_i remotely invokes a method of component c_j in order to send data to c_j or to receive data

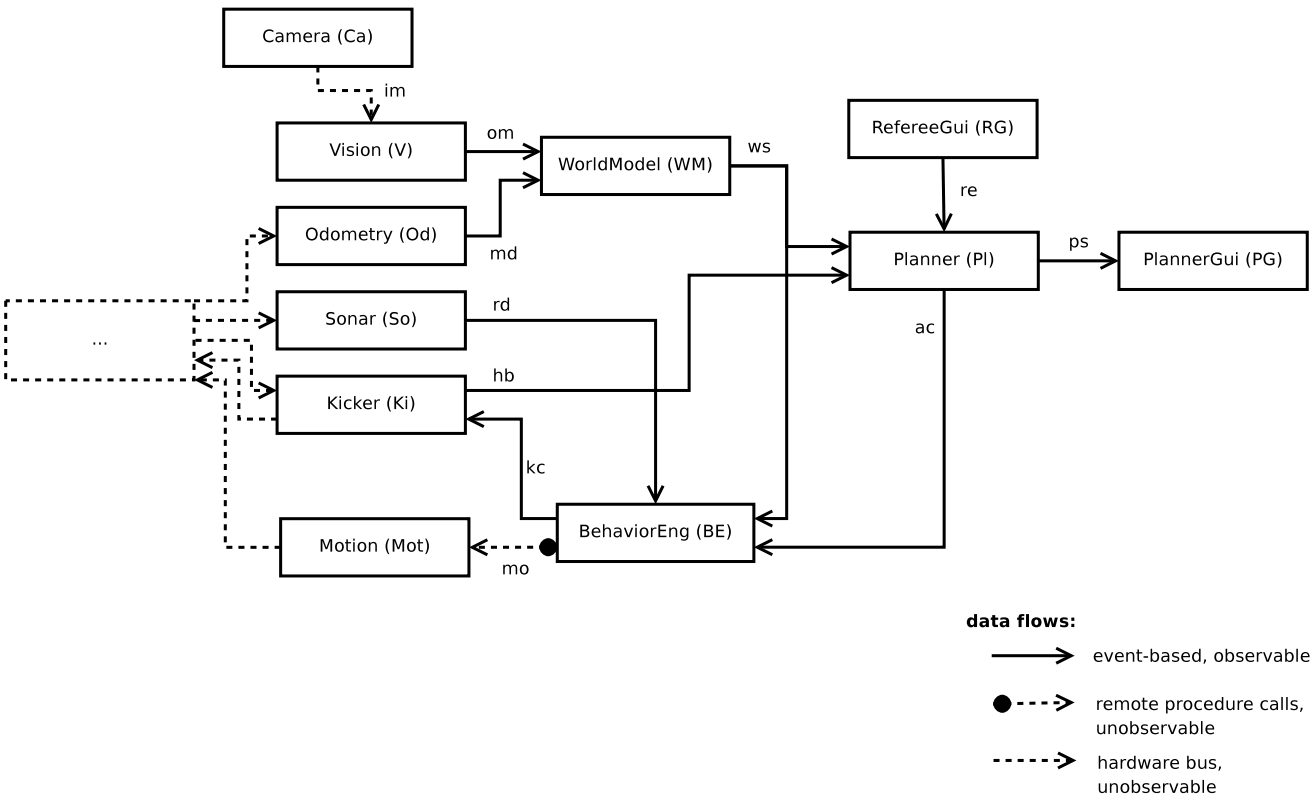


Fig. 2 Architectural view on the control system of an autonomous soccer robot. *Ca* is a hardware component, the other labelled rectangles represent software services

from c_j. Those software components which process sensor data are connected to hardware components over different kinds of hardware buses. We also distinguish between observable connections and unobservable ones. Data which is transmitted over the former can be easily intercepted from outside, while data on the latter is not observable by the diagnosis system.

The components operate concurrently. The components *V*, *Od*, *So* and *Ki* process low-level data received from hardware sensors and pass the processed sensor data on to other components. *WM* merges sensor data into a continuous world model which, among others, states the positions and movements of environment objects (e.g., other robots or soccer goals). *Pl* is an AI-planning system responsible for the high-level decision-making. It sends abstract actions to the component *BE*. *BE* implements the low-level behaviors which correspond to the high-level actions sent by the planner, and issues motion and kick commands to the components *Ki* and *Mot* which directly control the robot's actuators.

The abstract behavior model which we use for the runtime diagnosis assigns arbitrary *properties* to components and connections. Properties capture invariants of the system. For example, the component property (c, npr), where *npr* means "number of processes", expresses that the software component c must spawn a certain (minimum) number of processes (threads). When c crashes, then this property will be violated. An example for a connection property is (e, pe), where *pe* means "periodic events", indicating that a new event must occur on connection e at least every Δt_{max} milliseconds. A violation of this property could be, e.g., an indication of a deadlock somewhere in the system.

At runtime the diagnosis system continuously monitors the properties by employing *monitoring rules*. Monitoring rules are pieces of software which concurrently monitor the control system and detect property violations. They can be regarded as software plug-ins, i.e., one can implement arbitrary types of rules and integrate them into our diagnosis system. Each instance of a rule type refers to a specific component or connection. As an example, Algorithm 1 sketches a pseudocode for a rule which monitors the property (c, npr). An instance of this rule is created for each component c which has the (c, npr) property. Each rule instance is either violated or not violated. Moreover, the variable *violated* can also be accessed by the diagnosis engine. Another example for a rule is outlined in Algorithm 2; it evaluates the property (e, pe).

Violated rules indicate that one or more faults have occurred somewhere in the system. However, the detection of a fault is not sufficient for performing repair or reconfiguration actions, it is also necessary to locate the failed component(s). If only component properties like (c, npr) are violated, then the localization is trivial, as it is clear that c must

Algorithm 1: RULE_NUMBER_OF_PROCESSES(c, min)
Input: a component c and a threshold *min* which denotes a min. number of processes

```
(1) violated := false
(2) repeat forever:
(3)     n := number of processes currently spawned by c
(4)     if n < min then violated := true
```

Algorithm 2: RULE_PERIODIC_EVENTS($e, type, \Delta t_{max}$)
Input: a connection e, an event type, and a threshold Δt_{max}

```
(1) violated := false
(2) last_event_time := now()
(3) execute threads in parallel:
(4)     thread 1:
(5)         repeat forever:
(6)             if there is a new event of type at e
                    then last_event_time := now()
(7)     thread 2:
(8)         repeat forever:
(9)             if now() − last_event_time > Δt_max
                    then violated := true
```

have failed. However, if a connection property like (e, pe) is violated then it is not obvious which component is responsible, because the failure of a specific component may cause property violations at various connections in the system. Therefore, we utilize model-based diagnosis techniques for the fault localization. The system description *SD* (Definition 1) reflects the logical relationships between properties.

We explain this concept by means of an example. The components *V* and *Od* periodically produce new output data (notice that we assume here that the hardware works correctly). Moreover, the component *WM* generates a new output event for each event arriving at one of its two inputs, the connections *om* and *md*. Hence, if there are periodic events at these two connections, then this must also hold for the output connection ws. In the behavior model we use propositions of the form $ok(x, \varphi)$ which are true iff property φ holds for the component or connection x. We obtain the following behavior models which are parts of *SD*:

$$\begin{aligned}
&\neg ab(V) \rightarrow ok(om, pe)\\
&\neg ab(Od) \rightarrow ok(md, pe)\\
&\neg ab(WM) \wedge \big(ok(om, pe) \vee ok(md, pe)\big) \rightarrow ok(ws, pe) \quad (6)\\
&\ldots\\
&\text{and for every component } c\text{:} \quad \neg ab(c) \rightarrow ok(c, npr)
\end{aligned}$$

Intuitively, if $\neg ok(ws, pe)$ is observed, then the diagnosis algorithm will infer either that *WM* is abnormal or that both *V* and *Od* have failed. The diagnosis system obtains the set of observations, *OBS*, by querying the states of the monitoring rules. If a rule which is associated with a property

(x, φ) is violated, then $\neg ok(x, \varphi)$ is added to *OBS*; otherwise, $ok(x, \varphi)$ is added.

It should be noted that the models above are abstract because they only capture a part of the complete behavior of the components. For example, they only specify that certain events must be produced periodically, but the content (value) of the events is not considered. However, these models are often sufficient for detecting and locating severe faults like crashes or deadlocks. In [3] we presented more detailed models which, nevertheless, are still simple compared to the extremely complex behavior of real-world robot control systems.

After diagnosis the results are used to repair the system. In our application this is done by simply restarting or reseting the software components. In [1, 16] the authors presented first ideas and a prototypical application for a mobile robot which was based on a simple rule-based[1] system rather than model-based reasoning methods. The system observed the state of the robot using a monitoring approach, computed diagnoses, and was able to repair the control software of the mobile robot. The main disadvantage of the rule-based system was that in some cases the diagnoses were larger than necessary.

In summary the runtime diagnosis system comprises three parts. The first part is the monitoring system which produces observations *OBS*. From the observations and the system description *SD* the diagnosis engine computes diagnoses which explain the observed behavior. The diagnoses are fed into the repair engine which computes repair actions which are applied to the system under supervision, i.e., in our case the robot control system.

4 Dependent failures in a robot control system

The classical approaches to model-based diagnosis focus on minimal diagnoses. This strategy is well-founded if one assumes that components fail independently—an assumption which is usually made in model-based diagnosis. However, as De Kleer remarks in [17], dependent failures are quite common in some domains.

In component-oriented software systems common causes for dependent failures are communication channels which do not completely decouple the participants of the communication. Typical examples are remote procedure calls. In our robot control system (Fig. 2), component *BE* performs CORBA remote procedure calls to *Mot*. Therefore, when *Mot* fails, then the next remote call to *Mot* may fail as well (depending on the way in which *Mot* fails). Since *BE* is not able to properly handle such an error, it fails as well, i.e., it produces an error message and terminates.[2]

Now different scenarios are conceivable. First, if the outputs of *Mot* are not observable by the diagnosis system, then it may happen that only *BE* is identified as faulty. Notice that many real systems, in particular hardware-software hybrid systems, suffer from limited observability. This would result in a single minimal diagnosis $\Delta = \{BE\}$. Consequently, the diagnosis system tries to repair the system by restarting *BE*; but, as *Mot* is not yet recovered, *BE* will fail again immediately when attempting the next remote procedure call to *Mot*. The crucial point is that a repair which is solely based on minimal diagnoses may not suffice in case of dependent failures. Of course it might also happen that multiple components fail independently (almost) at the same time and that not all of these failures are recognized by the diagnosis system; however, multiple independent failures are extremely unlikely in most real systems, and if they occur then the affected components can usually be repaired independently from each other.

In order to tackle this problem, we propose the concept of *diagnosis environments* (*DEs*). DEs always refer to a specific diagnosis Δ, i.e., for each diagnosis there are one or more DEs. The DEs of Δ comprise all components in Δ and, in addition, they may contain further components which have caused the failure of components in Δ. As we will show in the next section, our approach provides the following two *minimal diagnosis environments* (*minimal DEs*) of $\Delta = \{BE\}$: $\theta_1 = \{indf(BE)\}$ and $\theta_2 = \{indf(Mot), df(Mot, BE)\}$, where the predicate *indf* denotes an independent failure and $df(Mot, BE)$ means that *BE* has failed in dependence of *Mot*. A repair based on these two minimal DEs is much more likely to succeed: even if an optimistic repair policy first tries to repair only *BE*, after the failed repair attempt it will consider θ_2 as the "right" minimal DE and restart both *Mot* and *BE*. This may finally result in a fully recovered system.

A second scenario is that the diagnosis system is able to identify both components as failed and thus compute the result $\Delta = \{Mot, BE\}$. Although this minimal diagnosis contains all components which have failed, the minimal DE $\theta = \{indf(Mot), df(Mot, BE)\}$ of Δ (note that there is only one minimal DE now) is much more helpful. First, the causal order of the failures also corresponds to the order in which the components should be repaired: clearly, *Mot* must be repaired before *BE*, because otherwise *BE* may fail dependently again. Second, the knowledge which of the components have failed dependently is often helpful per se: e.g.,

[1] In this context the term *rule* relates to `if..then..else` rules as used for reasoning in expert systems, in contrast to the monitoring rules as introduced above.

[2] Of course it would be desirable that every software component provides a high degree of fault-tolerance and thus may continue to operate even after such runtime errors. However, in many existing software systems this is not the case, and changing the design of legacy systems is often not possible or too costly.

it might help in the assessment of the reliability of components, or in later debugging sessions.

A similar example is a failure of the camera (component *Ca*). From our practical experience we know that a fault in the *Ca* often causes a crash of *Vi*, as *Vi* directly accesses the camera's hardware and thus is tightly coupled to *Ca*. Unfortunately, a failure of the camera cannot be directly detected from outside. Again, the minimal DEs $\theta_1 = \{indf(Ca), df(Ca, Vi)\}$ and $\theta_2 = \{indf(Ca)\}$ of the minimal diagnosis $\Delta = \{Vi\}$ provide better informations than Δ: after repeated restarts of *Vi*, the diagnosis system may consider to reset the camera hardware and restart *Vi* afterwards.

In our final example we show that components may fail dependently when they are based on a common framework. Our control system is based on CORBA, and for the asynchronous event-based communication among SW components (denoted by solid arrows in Fig. 2) we use a CORBA Event Channel (EC). For the sake of readability the component *EC*, which is connected to most other software components, is not contained in the model in Fig. 2. However, most of the SW components depend on *EC*: they post new events to *EC*, and/or they subscribe to events produced by other components. In our control system a failure of *EC* propagates to many other components, which often terminate (maybe after producing error messages). Again, due to limited observability this may result in a multiple-fault diagnosis which does not contain *EC*, and a repair based solely on minimal diagnoses will fail. Moreover, the independently failed component *EC* must be restored before the dependently failed components are restarted.

5 A general discussion on dependent failures

It is important to realize that dependent failures may also occur in many other domains. Consider the bulb example in Fig. 1. A short in a voltage source may lead to a very high current which eventually destroys the bulbs. If the magnitude of the voltage is not measured (note that this is a more "expensive" measurement than simply observing whether the lights are on or off), then the observations $OBS = \{fil_1 = fil_2 = broken\}$ will result in a single minimal diagnosis $\Delta = \{B_1, B_2\}$. However, when the bulbs are replaced without repairing the voltage source first, then the new bulbs will sooner or later be destroyed again. In this example our approach would, if the failure dependencies are properly modelled, come up with a minimal DE $\theta = \{indf(V), df(V, B_1), df(V, B_2)\}$.

A crucial question is whether a dependent failure should really be regarded as an *abnormal* behavior, because the dependent failure is caused by an unexpected and damaging input rather than an internal fault. E.g., one could adopt the view that a broken filament is a normal reaction to a high input voltage. This could be reflected in the following modification of the behavior model:

$$\neg ab(B_1) \wedge (u_{b1} \neq high) \rightarrow (fil_1 = intact) \tag{7}$$

where *high* denotes a voltage magnitude which is significantly higher than expected (in contrast to *pos* which denotes a "normal" voltage magnitude). Using this model, we could obtain two minimal diagnoses $\Delta_1 = \{V\}$ and $\Delta_2 = \{B_1, B_2\}$ for $OBS = \{fil_1 = fil_2 = broken\}$.

In most diagnosis applications the final purpose is a repair, hence every component which needs to be repaired in order to restore the full system functionality should be regarded as abnormal in the diagnoses. In other words, the results from the diagnostic process should directly allow for a repair of the entire system, and it should not be necessary that the repair procedure performs additional diagnostic activities. Note that the knowledge that *V* has failed is—per se—not sufficient to conclude that the bulbs have failed as well, because (in general) a component may fail independently in many different ways.

Moreover, even the observation $u_v = high$ does not necessarily imply that the bulbs have failed: all we know is that the lifespan of a filemant strongly decreases with higher voltage, and the abstract models which are typically used in model-based diagnosis often do not consider details like the exact voltage value, the period of time since u_v has become *high*, the age of the bulbs, etc. Hence, the model

$$(u_v = high) \rightarrow ab(V) \tag{8}$$

is not appropriate. It can be seen that the possible failure dependency between *V* and B_i is a *may* relationship which cannot be easily expressed in the system description *SD* (at least not if *SD* has the intented usage which is common to model-based diagnosis approaches). As we will see in the following section, we propose to model all possible failure dependencies in a separate dependency graph, and our approach provides logical diagnostic techniques which can determine whether one or both bulbs have actually failed dependently, taking the observations into account.

Moreover, suppose we certainly know that two components c_i and c_j have failed, and that a failure of c_i may cause a dependent failure of c_j. Only diagnostic reasoning can determine whether c_j has actually failed dependently or both of the components have failed independently. This knowledge can, e.g., help to optimize the repair procedure: in the latter case, it could be possible to reduce the repair time by repairing both components concurrently (depending on the application domain).

6 Diagnosis Environments (DEs)

As already mentioned, for each diagnosis there are one or more DEs. A DE of a diagnosis Δ contains all components

in Δ and, in addition, it may contain further components which are assumed to have caused a failure of any $c \in \Delta$. Thus, the computation of DEs is done after computing the minimal diagnoses of $SD \cup OBS$ (see Definitions 1 and 2) and might be regarded as "post-processing step". However, as we will see in Sect. 7.2, an interleaved computation of the minimal diagnoses and their minimal DEs should be preferred because it is often not necessary to compute all minimal diagnoses. A noteworthy strength of our approach is the fact that we can build upon Reiter's well-founded theory of diagnosis [5] and that approved algorithms can be applied for generating the minimal diagnoses. We formally extend this framework by re-defining the notion of a *system*. Remember that, in Reiter's framework, a system is a tuple $(SD, COMP, OBS)$; see Definition 1.

Definition 3 (System (extended definition)) A *system* is a tuple $(SD, COMP, FDG, DFD, OBS)$, where SD, $COMP$ and OBS have the same meaning as in Definition 1. The *failure dependency graph FDG* is a directed acyclic graph (DAG) which has a node for each component in the system and whose edges represent possible failure dependencies. *DFD* is the *dependent failure description* (*DFD*), as set of sentences modelling the behavior of dependently failed components.

The intended usage of *SD* is as usual in model-based diagnosis: using the literal $\neg ab(c)$, it contains sentences which specify the nominal behavior of components. Furthermore, we allow the usage of the fault mode $ab(c)$ in order to describe a faulty behavior. The sets *COMP* and *OBS* are used as usual.

Our first extension is the *failure dependency graph* (*FDG*). An edge from c_i to c_j in the FDG indicates that a failure of component c_i may also cause a failure of c_j. Note that, as a component c_i may fail in many possible ways, an edge does not state that c_j *must* fail. The FDG for our robot control system is depicted in Fig. 3. It can be seen that some components have multiple parents (= direct ancestors) in the FDG, i.e., there are several other components which may cause them to fail dependently.

Our second extension to the definition of a system, the dependent failure description (DFD), is explained later in this section.

While a diagnosis only distinguishes between abnormal and not abnormal components, a DE employs the already introduced literals $indf(c)$ and $df(c_i, c)$ to state the paths of failure propagation:

Definition 4 (Dependency modes) For each component there is a non-empty set of *dependency modes*. The set of available dependency modes is determined by the FDG: each component c has an $indf(c)$ mode and one mode $df(c_i, c)$ for each parent c_i of c in the FDG.

E.g., *Vi* has the dependency modes $\{indf(Vi), df(Ca, Vi), df(EC, Vi)\}$, whereas *EC* has no df-mode as it has no parents in the FDG. The literal $ab(c)$ is not used in DEs; i.e., each component considered as abnormal by a DE has an $indf$ or df mode. If a DE contains $df(c_i, c)$ then c_i must be abnormal as well, i.e., the DE must also contain either $indf(c_i)$ or $df(\cdot, c_i)$. The latter case indicates a cascade of failures: c_i fails due to another component and subsequently causes a dependent failure of c.

Suppose we have a minimal diagnosis $\Delta = \{Pl, BE\}$. Then, according to the known failure dependencies, we could obtain the following DEs for Δ:

1. $\theta_1 = \{indf(Pl), indf(BE)\}$
2. $\theta_2 = \{indf(EC), df(EC, Pl), indf(BE)\}$
3. $\theta_3 = \{indf(EC), df(EC, Pl), df(EC, BE)\}$
4. $\theta_4 = \{indf(Pl), indf(Mot), df(Mot, BE)\}$
5. $\theta_5 = \{indf(EC), df(EC, Pl), indf(Mot), df(Mot, BE)\}$

It can be seen that one may obtain a large number of DEs for a single diagnosis because the FDG is a static model stating all possible failure dependencies. Hence, we should seek to logically refute DE candidates which are implausible given the observations. We provide two possibilities for doing this. First, as already mentioned, we permit the specification of an abnormal behavior. E.g., if we assume that any failure of *EC* is a total failure, then we can say that no more events can be transmitted over those connections in the system which rely on the event channel:

$$ab(EC) \rightarrow \neg ok(om, pe) \wedge \neg ok(md, pe) \wedge \ldots \quad (9)$$

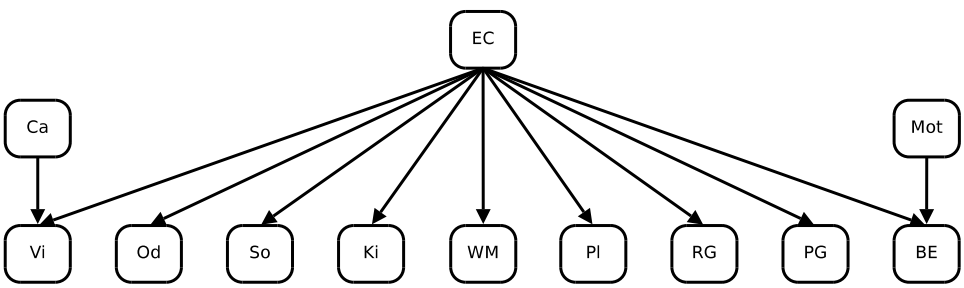


Fig. 3 Failure dependency graph (FDG) for the control system in Fig 2. *EC* is the CORBA event channel

Therefore, if the property (e, pe) still holds for any of those connections e, then EC must not be abnormal. Clearly, if EC cannot be abnormal, there should be no DE which contains $indf(EC)$. More generally, it is necessary to establish a logical relationship between the notions of *abnormality* and (*in-*)*dependent failure*. This is achieved by the following definition:

Definition 5 (Dependency Mode Axioms (DMA)) The set DMA comprises the following axioms:

$$\begin{aligned} &\forall c \in COMP, \quad indf(c) \rightarrow ab(c) \\ &\forall c_i, c_j \in COMP, \quad df(c_i, c_j) \rightarrow ab(c_i) \wedge ab(c_j) \end{aligned} \tag{10}$$

Hence, every independently or dependently failed component is abnormal; in other words, the abnormality of a component is the consequence of its independent or dependent failure. Intuitively, it follows that, if the assumption $ab(EC)$ leads to a logical contradiction, $indf(EC)$, cannot be included in any DE.

As a second possibility to refute implausible DEs we introduced the *dependent failure description* (DFD), which is a part of the system model as indicated in Definition 3. The intended usage of DFD is to model the behavior of dependently failed components by sentences of the form

$$df(c_i, c_j) \rightarrow \ldots \tag{11}$$

for example

$$df(Mot, BE) \rightarrow \neg ok(BE, cerr) \tag{12}$$

where $ok(BE, cerr)$ holds only if BE has *not* reported any CORBA error.[3] This fault model states that, if BE fails in dependence of Mot (due to a failed CORBA remote procedure call), then BE issues a CORBA error. Thus, if BE fails without reporting such an error, then there cannot be any DE containing $df(Mot, BE)$, as it would contradict the observations.

Now we have the necessary prerequisites to define the DEs of a (minimal or not minimal) diagnosis Δ. Note that DEs are also defined for non-minimal DEs, even though in practice one will usually only be interested in the DEs of minimal diagnoses.

Definition 6 (Diagnosis Environment (DE)) Let $\Gamma(\theta)$ denote the set of all components c with either $indf(c) \in \theta$ or $df(\cdot, c) \in \theta$, and $\overline{\Gamma}(\theta) = COMP \setminus \Gamma(\theta)$. Then each diagnosis environment θ of Δ is an assignment of dependency modes (Definition 4) to components s.t.:

1. $SD \cup DFD \cup DMA \cup OBS \cup \theta \cup \{\neg ab(c) \mid c \in \overline{\Gamma}(\theta)\}$ is consistent.
2. For all components $c \in \Delta$: $c \in \Gamma(\theta)$.
3. For all $c_{i_1} \in \Gamma(\theta)$: either $c_{i_1} \in \Delta$ or there is a component $c_{i_k} \in \Delta$ s.t.:
   $\{df(c_{i_1}, c_{i_2}), \ldots, df(c_{i_{k-1}}, c_{i_k})\} \subset \theta$ with $k \geq 2$.

Remember that SD, DFD, and OBS are parts of the system (Definition 3), and DMA was defined in Definition 5. Item 1 in Definition 6 means that θ, together with the assumption that all components which are not in θ behave normally, must be consistent with $SD \cup DFD \cup DMA \cup OBS$. This definition is similar to the definition of a diagnosis (Definition 2). DFD and DMA (the dependent failure description and the dependency mode axioms, as introduced above) are our extensions. The DMAs are required for relating the dependency modes to the ab fault mode.

Item 2 in Definition 6 says that every component which is in a diagnosis must also be assumed as failed in all DEs of this diagnosis. Finally, Item 3 ensures that only those components c_{i_1} are in θ which are either in Δ or which may have (directly or indirectly) caused a failure of any component $c_{i_k} \in \Delta$. E.g., consider again the example $\Delta = \{Pl, BE\}$. All of those 5 DEs, which are shown above, contain Pl and BE. In addition, some of them also assume EC and/or Mot as failed. However, none of those DEs contains, say, Od, because this would not conform to Item 3.

As already explained, the ab fault models in SD and the fault models in DFD allow us to logically refute implausible DEs. However, in practice it is often the case that the fault models are incomplete, i.e., that the faulty behavior of components is only partially known. As a consequence it is still possible that a diagnosis has a large number of DEs. Hence, we propose to focus on *minimal diagnosis environments*:

Definition 7 (Minimal Diagnosis Environment (minimal DE)) Let $\#(\theta)$ denote the number of $indf$ modes in a DE θ. For a given system, a DE θ is *minimal* iff there is no other DE θ' with $\#(\theta') < \#(\theta)$.

The intuition behind this focusing strategy is the insight that, in most practical systems, multiple independent failures are very unlikely. For example, the diagnosis $\Delta = \{Pl, BE\}$ has only one minimal DE $\theta = \{indf(EC), df(EC, Pl), df(EC, BE)\}$, provided that θ also conforms to Item 1 of Definition 6.

In general we obtain several minimal diagnoses for a system, and therefore we extend the concept of minimal DEs to a set of minimal diagnoses:

Definition 8 ((Minimal) DEs of a system—$\Theta(\mathfrak{D})$) Let $\Theta(\Delta)$ be the set of all DEs of a minimal diagnosis Δ, and

[3] We assume that components may produce error messages which are observable by the diagnosis system.

let $\mathfrak{D}$ be the set of all minimal diagnoses for a system. Then $\Theta(\mathfrak{D})$ denotes all DEs of this system:

$$\Theta(\mathfrak{D}) = \bigcup_{\Delta \in \mathfrak{D}} \Theta(\Delta) \tag{13}$$

Moreover, $\theta \in \Theta(\mathfrak{D})$ is minimal in $\Theta(\mathfrak{D})$ iff $\#(\theta) \leq \#(\theta')$ for all $\theta' \in \Theta(\mathfrak{D})$. We say that θ is a minimal DE of the system.

Finally, before presenting algorithms for the computation of all minimal DEs of a system, we briefly outline some theoretical results which follow from our definition of DEs (please refer to [9] for details):

- If $\Delta = \{c_{i_1}, \ldots, c_{i_m}\}$ is a diagnosis then $\theta = \{indf(c_{i_1}), \ldots, indf(c_{i_m})\}$ is a DE of Δ. The intuition behind is that, if a component behaves abnormally, then it is always possible that it has failed independently. From this proposition it follows that every diagnosis has at least one DE.
- If $\Delta = \{c_{i_1}, \ldots, c_{i_m}\}$ is a diagnosis and the FDG has no edges (i.e., there are no failure dependencies), then Δ has exactly one DE: $\theta = \{indf(c_{i_1}), \ldots, indf(c_{i_m})\}$.
- If Δ is an empty diagnosis, i.e., $\Delta = \{\}$, then it has exactly one DE: $\theta = \{\}$.
- If Δ is a non-empty diagnosis, then every DE of Δ has at least one *indf* mode, but the number of *indf* modes cannot be greater than $|\Delta|$.

7 Computing the minimal diagnosis environments

7.1 Simple algorithm

We first present a simple algorithm which computes all minimal DEs for a system, and describe important improvements in the following section.

The algorithm is depicted in Algorithm 3. If there is no empty diagnosis, then it computes all minimal diagnoses (using, e.g., Reiter's Hitting Set algorithm) and creates a *diagnosis environment tree* (*DE-tree*) T_Δ for each minimal diagnosis Δ. Each node n of a DE-tree represents a mode assignment, denoted by $n.\theta$, which is a DE candidate. More precisely, $n.\theta$ is a DE iff the consistency check (as shown in the algorithm) returns *true*. The algorithm first attempts to generate DEs with only one *indf* mode. If such DEs are found, then they compose the minimal DEs of the system. Otherwise, it is also required to perform consistency checks for nodes with two *indf* modes, etc.

Figure 5 depicts a DE-tree for the failure dependency graph (FDG) in Fig. 4. This DE-tree refers to the minimal diagnosis $\Delta = \{C, E\}$. The root node of any DE-tree T_Δ consists of an *indf* mode for all $c \in \Delta$. Algorithm 4 depicts a pseudocode for creating the children of a node.

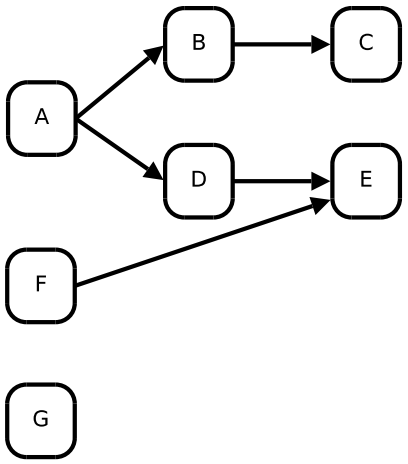


Fig. 4 Failure dependence graph (FDG) of an example system

Algorithm 3: Compute all minimal DEs of a system (simple algorithm)
Input: A system (*SD*, *COMP*, *FDG*, *DFD*, *OBS*) (see Definition 3)
Output: All minimal DEs of a system, denoted by $\Theta(\mathfrak{D})$

(1) compute $\mathfrak{D}$, the set of all minimal diagnoses of the system
(2) **if** there is an empty diagnosis $\Delta = \{\}$
  then return $\Theta(\mathfrak{D}) = \{\}$
(3) **else**
(4) **for all** $\Delta \in \mathfrak{D}$: generate the complete *diagnosis environment tree* T_Δ
(5) $\Theta(\mathfrak{D}) := \{\}$; $\nu := 1$
(6) **while** $\Theta(\mathfrak{D})$ is empty:
(7) **for all** $\Delta \in \mathfrak{D}$, **for all** nodes n in T_Δ with $\#(n.\theta) = \nu$:
(8) **if** $SD \cup DFD \cup DMA \cup OBS \cup n.\theta \cup \{\neg ab(c) | c \in \overline{\Gamma}(n.\theta)\}$ is consistent
  then $\Theta(\mathfrak{D}) := \Theta(\mathfrak{D}) \cup \{n.\theta\}$
(10) **end**
(11) $\nu := \nu + 1$
(12) **end**
(13) **return** $\Theta(\mathfrak{D})$
(14) **end**

It is important to note that each DE-tree should contain only unique DE candidates in order to reduce the computational costs. For example, in Fig. 5 it can be seen that the application of Algorithm 4 to node n_7 creates a child n' (denoted by "[4]" in the figure) which is equal to node n_4. Obviously n' is redundant as all of its possible descendants are also descendants of n_4, and thus n' is not added to the DE-tree. The important observation is that the mode $indf(B)$ in n_2 is replaced by $\{indf(A), df(A, B)\}$ in n_3, and that the mode $indf(B)$ in the descendants of n_7 and n_9 (more general, in the descendants of the siblings of n_3) therefore never

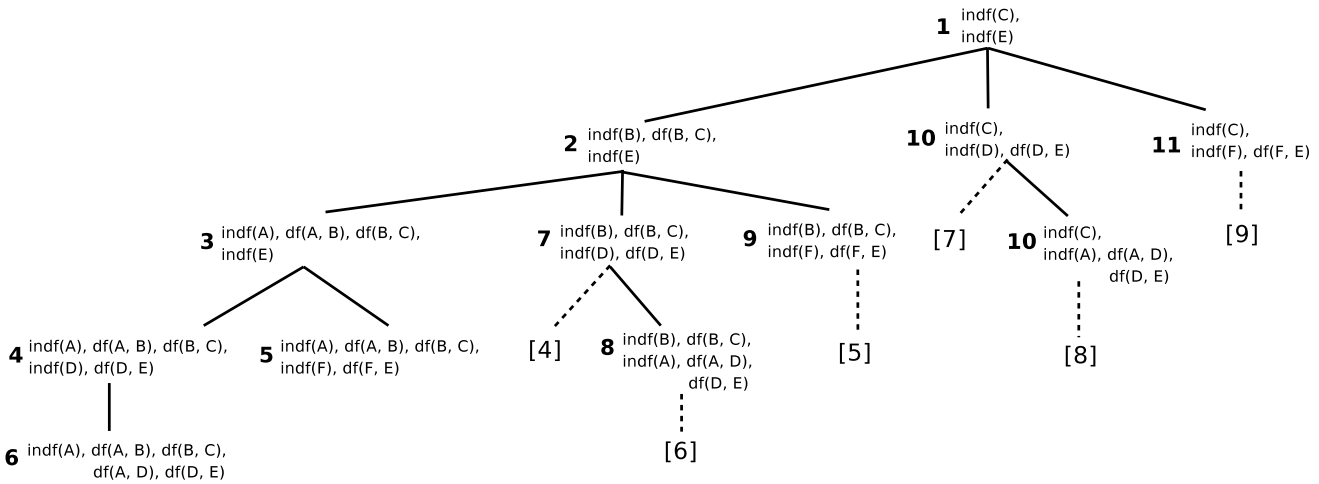


Fig. 5 The diagnosis environment tree (DE-tree) T_Δ for a minimal diagnosis $\Delta = \{C, E\}$ w.r.t. the FDG depicted in Fig. 4. Each node n_i is labelled with the number i. A redundant node which is actually not added to the tree because it would be equal to another node n_k is denoted by the notation $[k]$

Algorithm 4: Expand a node n
Input: a node n of a DE-tree and the failure dependency graph *FDG*
Output: a set N' containing the children of n
(1) $N' := \{\}$
(2) **for all** c with $indf(c) \in n.\theta$:
(3) **for all** c_i which are parents of c in *FDG*:
(4) create a new node n' with $n'.\theta := n.\theta$
(5) **if** neither $indf(c_i) \in n'.\theta$ nor $df(\cdot, c_i) \in n'.\theta$
then $n'.\theta := n'.\theta \cup \{indf(c_i)\}$
(7) $n'.\theta := \big(n'.\theta \setminus \{indf(c)\}\big) \cup \{df(c_i, c)\}$
[i.e., substitute $df(c_i, c)$ for $indf(c)$ in $n'.\theta$]
(9) $N' := N' \cup \{n'\}$
(10) **end**
(11) **end**
(12) **return** N'

needs to be replaced by $\{indf(A), df(A, B)\}$ as the resulting DE candidates are already in the subtree whose root is n_3. Hence, the nodes n_7 and n_9 and all descendants of them should contain a tag stating that $indf(B)$ does not need to be substituted. This idea can be straightforwardly generalized to all nodes in a DE-tree; we omit a more detailed description here.

In this example there is only one DE candidate with a single *indf* mode, namely n_6. If this node is not a DE (because the consistency check is negative) then all minimal DEs of the system have two *indf* modes. Moreover, the fact that the root node of a DE-tree is always a DE (see [9] for details) proves that Algorithm 3 always terminates. Finally, note that component G is not included in any DE of Δ.

7.2 Improved algorithm

A severe drawback of the algorithm presented in Sect. 7.1 is that it requires the generation of *all* minimal diagnoses and that it always creates the complete DE-tree for each of them. In this section we outline some improvements which reduce the computational costs of generating the minimal DEs of a system.

Considering the FDG in Fig. 4, we observe that any possible node of a DE-tree contains the modes $indf(C)$ and $indf(E)$ has a descendant node with only one *indf* mode, namely $indf(A)$. The reason is that A is a common ancestor of C and E in the FDG. On the other hand, all descendants of a DE-tree node which contain $indf(C)$ and $indf(F)$ have two *indf* modes, as C and F have no common ancestor in the FDG.

This observation motivates an important improvement of the algorithm: the algorithm should first focus on DE candidates with a single *indf* mode, and only if no such DE is found it should consider DE candidates with two *indf* modes. It follows that a node which contains, e.g., $indf(C)$ and $indf(F)$ should not be expanded before it is clear that no minimal DE with only one *indf* mode exists. In many cases this will significantly reduce the search space. In Fig. 5 the nodes n_9 and n_{11} would tentatively remain unexpanded; coincidentally, in this example the descendants of the affected nodes are redundant anyway.

We introduce a function $\#_{\geq}(n)$ which returns a lower boundary for the number of *indf* modes of a node n and all of its descendants in the DE-tree. Trivially, this function could always return 1, but we seek a better estimation. If n contains only two *indf* modes, $indf(c_i)$ and $indf(c_j)$, then $\#_{\geq}(n) = 1$ if there is a directed path in the FDG from c_i

to c_j (or vice versa) or if there is a component c_k s.t. there are directed paths in *FDG* from c_k to both c_i and c_j; otherwise $\#_{\geq}(n) = 2$. This idea can be generalized to $\#(n.\theta) = k$ with $k \geq 2$; we omit the details.

Of course, the considerations above also apply to the root nodes of DE-trees. As a root node directly stems from a minimal diagnosis we make another observation. Suppose, for example, that we have two minimal diagnoses, $\Delta_1 = \{C, E\}$ and $\Delta_2 = \{C, F\}$. Then the corresponding root nodes of the two DE-trees would be $n_{r1} = \{indf(C), indf(E)\}$ and $n_{r2} = \{indf(C), indf(F)\}$, respectively. Clearly, $\#_{\geq}(n_{r1}) = 1$ and $\#_{\geq}(n_{r2}) = 2$. Hence, if Δ_1 has a DE with only one *indf* mode, then this is a minimal DE of the system, and therefore no DE of Δ_2 can be a minimal DE of the system since all of those DEs must have two *indf* modes.

Let $\#_{\geq}(\Delta)$ be an abbreviation for $\#_{\geq}(n_r)$ where n_r is the root node corresponding to Δ. The crucial observation is that, if we want to compute the minimal DEs of a system, a minimal diagnosis Δ with $\#_{\geq}(\Delta) = 2$ does not even need to be generated if there are minimal DEs with only one *indf* mode. Hence, the improved algorithm should first focus on minimal diagnoses Δ with $\#_{\geq}(\Delta) = 1$. Notice that Reiter's hitting set algorithm can be easily extended to compute only those minimal diagnoses in the first step, and to compute further minimal diagnoses if necessary.

The resulting algorithm for computing all minimal DEs of a system is sketched in Algorithm 5. We use $G = \{T_{\Delta 1}, \ldots, T_{\Delta m}\}$ to denote a *diagnosis environment graph* which consists of m DE-trees (for m different minimal diagnoses).

A further improvement can be achieved by utilizing *conflict sets* [5, 18]. Intuitively, a conflict set is a set of mode assumptions which are inconsistent with $SD \cup DFD \cup DMA \cup OBS$. For example, the conflict $\gamma = \{indf(A), df(B, C)\}$ states that either $\neg indf(A)$ or $\neg df(B, C)$ holds. Hence, the conflict can also be represented by the clause $\neg indf(A) \vee \neg df(B, C)$. If the reasoner which performs the consistency check is able to return a conflict when a DE candidate is logically refuted, those conflicts can be utilized to prune the search space. For example, if the consistency check for the node n_3 in Fig. 5 returns the conflict γ, then it is immediately clear that any descendant n' of n_3 must be inconsistent as well because $\gamma \subset n'.\theta$ holds. A more detailed discussion of this issue is provided in [9].

Algorithm 5: Compute all minimal DEs of a system (improved algorithm)
Input: A system (*SD*, *COMP*, *FDG*, *DFD*, *OBS*)
Output: All minimal DEs of a system, denoted by $\Theta(\mathfrak{D})$

(1) **if** there is an empty diagnosis $\Delta = \{\}$
  then return $\Theta(\mathfrak{D}) = \{\}$
(2) **else**
(3) $\Theta(\mathfrak{D}) := \{\}$; $\nu := 1$; $G = \{\}$
(4) **while** $\Theta(\mathfrak{D})$ is empty:
(5) compute all minimal diagnoses Δ with $\#_{\geq}(\Delta) = \nu$, store them in a set $\mathfrak{D}_\nu$
(6) **for all** $\Delta \in \mathfrak{D}_\nu$:
(7) create an initially empty DE-tree T_Δ
(8) create the corresponding root node for Δ, add it to T_Δ
(9) $G := G \cup \{T_\Delta\}$
(10) **end**
(11) **repeat**
(12) select an unexpanded node n from any $T_\Delta \in G$ with $\#_{\geq}(n) = \nu$
(13) expand n (see Algorithm 4)
(14) **until** no more nodes can be selected
(15) **for all** nodes n in any $T_\Delta \in G$ with $\#(n.\theta) = \nu$:
(16) **if** $SD \cup DFD \cup DMA \cup OBS \cup n.\theta \cup \{\neg ab(c) \mid c \in \overline{\Gamma}(n.\theta)\}$ is consistent
  then $\Theta(\mathfrak{D}) := \Theta(\mathfrak{D}) \cup \{n.\theta\}$
(18) **end**
(19) $\nu := \nu + 1$
(20) **end**
(21) **return** $\Theta(\mathfrak{D})$
(22) **end**

8 Repair in systems with dependent failures

As already discussed, the issue of dependent failures has a strong impact on repair policies. In many systems, the causal order of the component failures corresponds to the order in which the components should be repaired. I.e., if a component c_j has failed in dependence of c_i, then it may be necessary to restore c_i before repairing c_j, because otherwise the faulty behavior of c_i could again propagate to the repaired c_j and hence lead to another dependent failure of c_j.

We attempt to restore a partly failed control system by restarting the affected components. In general we do not obtain a unique minimal DE, i.e., a repair strategy must be able to deal with a set of minimal DEs which may contain components which actually have not failed in reality. Furthermore, repair strategies are always influenced by the attributes of the target system. We propose an algorithm which is tailored to software systems which have the characteristics that the repair (= restart) of a component is cheap w.r.t. the consumed resources: the sole costs caused by a restart is the time required for the restart.

Consequently, we choose a *pessimistic* repair strategy which presumes that *all* components which occur in any minimal DE of the system have failed and thus need to be restarted. Moreover, the order in which the components are repaired corresponds to the strictest order which is indicated by any minimal DE. We explain this by means of an example. Suppose there are three minimal DEs of a system: $\theta_1 = \{indf(U), df(U, V), df(V, W), df(V, X)\}$, $\theta_2 = \{indf(V),$

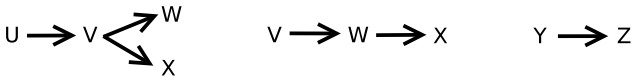
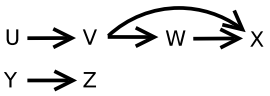


Fig. 6 The graphical representation of the minimal DEs θ_1, θ_2 and θ_3 as directed acyclic graphs

Fig. 7 The repair graph obtained by merging the minimal DEs θ_1, θ_2 and θ_3



Algorithm 6: Repair algorithm
Input: A set of minimal DEs $\Theta(\mathfrak{D}) = \{\theta_1, \ldots, \theta_m\}$
(1) abort all components occurring in $\Theta(\mathfrak{D})$
(2) generate the repair graph R
(3) **repeat**
(4) select any aborted component c which either has no parent in R or which has only parents which have already been restarted
(5) restart c in a separate process (i.e., restarts are executed concurrently with this repair algorithm)
(6) **until** all components in $\Theta(\mathfrak{D})$ have been restarted

$df(V, W)$, $df(W, X)\}$ and $\theta_3 = \{indf(Y)$, $df(Y, Z)\}$. Figure 6 depicts the graphical representations of those minimal DEs as directed acyclic graphs (DAGs). We see that, according to θ_1, the components W and X could be repaired in parallel, whereas θ_2 indicates that W must be repaired before X. Our pessimistic strategy chooses the stricter order imposed by θ_2. Note that in many systems there may be additional constraints on the repair order and schedule which are not related to the dependencies of failures.

In order to determine the order in which the components are repaired our algorithm creates a *repair graph*. It is a directed acyclic graph which results from merging the minimal DEs of the system. Regarding the obtained minimal DEs as directed acyclic graphs, a repair graph is the union of those graphs, i.e., it contains every component and every edge between two components occurring in any minimal DE. Figure 7 shows the repair graph for our example, and Algorithm 6 outlines the overall repair algorithm.

Note that this algorithm is simplified in the sense that it assumes that all components can be restarted successfully. In practice it may happen that restarted components immediately fail again. The algorithm we implemented addresses this problem by removing repeatedly failed components from the system. In such a case the behavior model of this component is also removed from the system description *SD*, and the AI-planning system which makes the high-level decisions of the robot is notified about the decreased capabilities of the control system; see [4] for details. Furthermore, notice that the diagnosis system suspends the monitoring during the repair.

9 Case studies

We implemented the proposed runtime monitoring and repair system in C++. The diagnosis engine, which computes the minimal diagnosis (using Reiter's Hitting Set algorithm) and the minimal DEs, was written in Java. It also includes a propositional Horn clause theorem prover for the consistency checks [19]. The reason why we encoded our models as Horn clauses is that they are amenable to efficient logical reasoning: the complexity of consistency checks is $O(n)$ where n is the number of literals. The monitoring system communicates with the diagnosis engine over TCP connections, using a text-based protocol. Moreover, as the behavior models are represented by XML files and the monitoring rules are software plug-ins, the whole diagnosis system is independent from a particular robot control system.

The case studies we present here were conducted on the software system which is depicted in Fig. 8. It is very similar to the control system in Fig. 2. The system was running on a PC, and the robot hardware was simulated using a soccer robot simulator (component *Sim*). The components *WM*, *Pl*, *RG*, *PG*, and *BE*, as well as the underlying framework based on CORBA, are very complex software services which have been used in real soccer robot tournaments (RoboCup Middle-size league). The control system comprises, even without the CORBA-based framework, more than 100,000 lines of code, and the framework is even much more complex. As our approach focuses on robot control systems rather than hardware issues we decided to perform these experiments offline relying on a hardware simulator. This setting provides a high degree of flexibility and allowed us to conduct more interesting and complex experiments.

The system comprises 11 components, including the *EventChannel* (*EC*) which is not shown in Fig. 8. The failure dependency graph (FDG) is depicted in Fig. 9. The system description *SD* contains about 230 Horn clauses, the dependent failure description *DFD* has about 18 clauses, and we typically obtain some 40 clauses for the observations *OBS* (one for each property). The most commonly used properties are the component property (c, npr) ("min. number of processes") and the connection property (e, pe) ("periodic events"), as introduced in Sect. 3, and the component property $(c, cerr)$ (holds as long as no CORBA error occurs) as used in (12) in Sect. 6. The software was running on a laptop with a 2 GHz Pentium M CPU and 1.5 GB RAM. The operating system was Linux.

9.1 Scenario 1

We conducted a very challenging experiment in which we injected a fault into the *EC* component which causes a total crash of the event channel *EC*. As discussed in Sect. 4, in

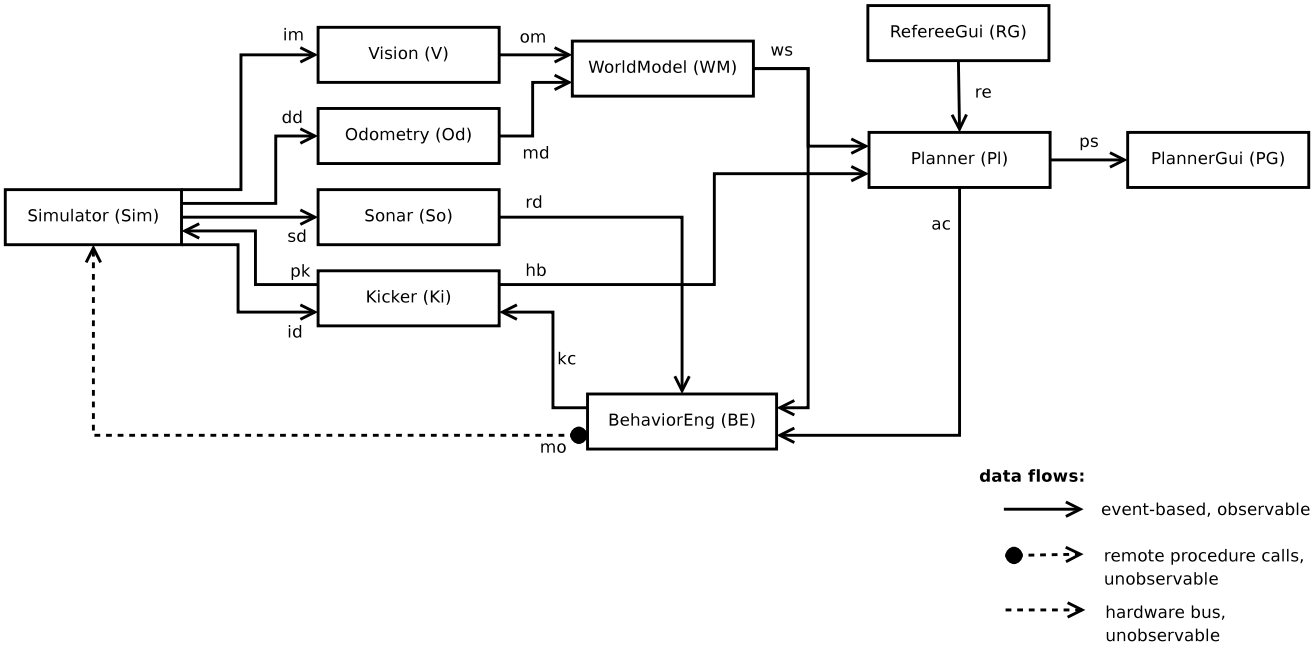


Fig. 8 Architecture of the software system used in the case studies

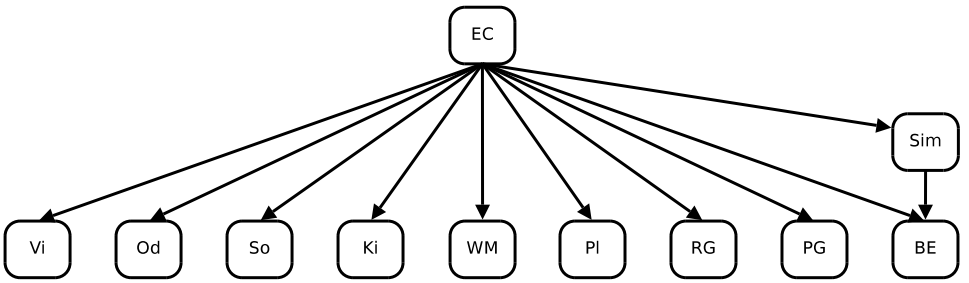


Fig. 9 Failure dependency graph (FDG) for the system in Fig. 8

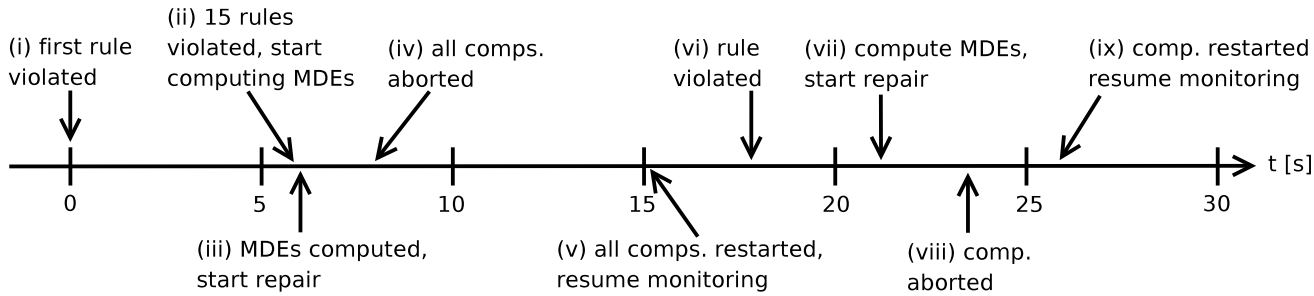


Fig. 10 Timeline for the first scenario

such a situation all components which send and/or receive events over the event channel may fail dependently.

Figure 10 depicts a timeline for this scenario. After the fault injection, the first property violation was detected within less than 800 ms, denoted by (i) in the figure. Then the diagnosis system continued the monitoring for about 6 s in order to be able to detect further violations. Intuitively, the diagnostic precision increases when more failure symptoms can be gathered. After this period the minimal diagnoses and the minimal DEs were computed for $OBS = \{\neg ok(EC, npr),\ \neg ok(om, pe),\ \neg ok(md, pe),\ \neg ok(ws, pe),\ ok(WM, npr),\ ok(Pl, npr),\ \neg ok(Pl, cerr), ok(BE, npr, \ldots\}$. It can be seen that some components, like *BE*, crashed (i.e., they lost all of their processes), while other failed components, e.g. *Pl*, did not crash but they reported CORBA errors.

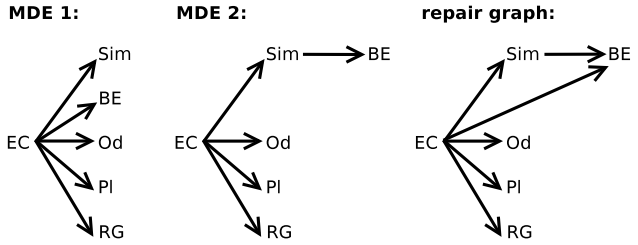


Fig. 11 The two minimal DEs obtained in this scenario for $\Delta = \{EC, Sim, Od, Pl, RG, BE\}$, and the resulting repair graph

The latter components must also be repaired as they are in a failed state from which they cannot recover.

The diagnosis system obtained a single minimal diagnosis $\Delta = \{EC, Sim, Od, Pl, RG, BE\}$, and two minimal DEs $\theta_1 = \{indf(EC), df(EC, Sim), df(EC, BE), df(EC, Od), df(EC, Pl), df(EC, RG)\}$ and $\theta_2 = \{indf(EC), df(EC, Sim), df(Sim, BE), df(EC, Od), df(EC, Pl), df(EC, RG)\}$. They are depicted in Fig. 11. Then the repair started and took about 9 s (the end of this period is denoted by (v) in the timeline). *EC* was restarted first, and *BE* only after *EC* and *Sim* were restored.

The problem in this scenario was that *WM* and *PG* also failed, but they did not exhibit explicit symptoms and thus were not included in the minimal DEs. However, after the monitoring was resumed a violation of the property (ws, pe) was detected ((vi) in Fig. 10), and the diagnosis system finally obtained the minimal DE $\{indf(WM)\}$ and started a second repair session (vii). *WM* could be restarted successfully, but the failure of *PG* was never diagnosed due to limited observability.

Nevertheless, our approach was able to detect and locate most of the failures and to correctly determine their dependencies. Within approximately 27 s after the injection of the fault in *EC* most of the control system functionality was restored without any human intervention; only a non-vital component remained faulty. The crucial point is that the minimal DEs provided the necessary knowledge about the order in which the components must be repaired, and even if *EC* were not a part of the minimal diagnosis (this would have happened, e.g., if we injected a deadlock instead of a total crash) our approach would have been able to identify *EC* as the independently failed component.

Also note that the computational overhead imposed by the continuous monitoring of the system was negligible ($< 5\%$ CPU usage), and that the computation of the minimal diagnoses and the minimal DEs together required less than 30 ms. This is also due to the small size of the model.

It can be seen that the time span between the detection of the first property violation and the start of the repair depends almost solely on how long the monitoring continues after the detection of the violation. In our case this period is 3–6 s. There are two reasons why we chose such a long time span. First, this increases the diagnostic precision as more failure symptoms can be observed. In applications in which software failures may immediately cause serious harm the time span may be reduced at the price of lower precision; i.e., it might happen that the repair starts before all failures can be diagnosed. Second, the length of the time span depends on the time granularity of the properties in the system model. E.g., the component *Pl* periodically produces new events at connection *ps*, and so we defined a property (ps, pe). The average period is about 1000 ms; however, this system is not designed as a hard real time system, and thus the actual periods may vary, in particular when the CPU load is heavy. We observed that applying too restrictive thresholds, e.g., 1200 ms for the property (ps, pe), may lead to false alarms, and so we defined more tolerant thresholds for the monitoring rules (e.g., 2500 ms for (ps, pe)). The important point is that in systems with lower time granularities the time span between the first detection of a fault and the start of the repair may be much smaller than in our example system.

9.2 Scenario 2

In the previous scenario the dependent failure of some components could not be identified by the diagnosis system. A closer look at the software system reveals that, while some components have a certain degree of fault tolerance and thus may survive the failure of *EC* (i.e., they suspend their operation while *EC* is not available and may later be able to reconnect to *EC* when it is restored), other components like *WM* and *PG* always fail in dependence of *EC*. These insights can be easily captured in the model: we added

$$\begin{aligned} ab(EC) &\rightarrow ab(WM) \\ ab(EC) &\rightarrow ab(PG) \end{aligned} \tag{14}$$

to the system description *SD*.

Again, we injected a fault in *EC*. This time we obtained the minimal diagnosis $\Delta = \{EC, Sim, WM, Pl, PG, RG, BE\}$ and the two minimal DEs $\theta_1 = \{indf(EC), df(EC, Sim), df(EC, WM), df(EC, Pl), df(EC, PG), df(EC, RG), df(EC, BE)\}$ and $\theta_2 = \{indf(EC), df(EC, Sim), df(Sim, BE), df(EC, WM), df(EC, Pl), df(EC, PG), df(EC, RG)\}$. In this scenario the minimal DEs contained all of the actually failed components. The system automatically restarted all of them. During the repair also the component *Ki* failed during an attempt to reconnect to *EC*. The diagnosis system was able to detect the failure after the first repair session and computed the minimal DE $\{indf(Ki)\}$. After the restart of *Ki* the software system was completely restored and fully operational. The timespan between the fault injection and the full recovery of the system was about 26 s.

9.3 Scenario 3

We simulated a multiple independent failure by killing the processes of *Pl* and *WM* almost simultaneously. The following properties were violated: (*Pl*, *npr*), (*ps*, *pe*), and (*WM*, *npr*). Hence, there was a single minimal diagnosis $\Delta = \{Pl, WM\}$, and our approach was able to correctly determine that both components have failed independently: we obtained only one minimal DE $\theta = \{indf(Pl), indf(WM)\}$.

The important point in this scenario is that the DE candidate $\{indf(EC), df(EC, Pl), df(EC, WM)\}$ was logically refuted due to the following fault models:

$$\begin{aligned} &ab(EC) \rightarrow \neg ok(om, pe) \wedge \neg ok(md, pe) \wedge \ldots \\ &df(EC, Pl) \rightarrow \neg ok(Pl, cerr) \end{aligned} \tag{15}$$

As *Pl* did not report any CORBA error, *OBS* contained *ok*(*Pl*, *cerr*), and thus it was impossible that *Pl* had failed in dependence of *EC*. Furthermore, the assumption *ab*(*EC*) led to a contradiction as properties like (*om*, *pe*) were not violated.

Our diagnosis system was able to completely restore the system within less than 11 s after the two component failures.

10 Related research

There is little related research which deals with model-based runtime diagnosis of software systems, and we are not aware of any work by other researchers which specifically focuses on the diagnosis and repair of robot control systems. In [20] an approach for model-based monitoring of component-based software systems is described, and the author of [21] proposes a model-based approach for the fault localization in Web Services. Apart from other differences, those works do not address the issues of dependent failures.

The authors of [22, 23] present research in the area of model-based software debugging. The aim of these works is the verification of fault localization at compile time. Another approach to the modelling of a software behavior is presented in [24]. In order to deal with the complexity of software, the authors propose to use probabilistic, hierarchical, constraint-based automata. However, they model the software in order to detect faults in hardware. Software bugs are not considered in this work. Furthermore, there are model-based approaches in the field of autonomic computing which aim at the creation of self-healing and self-adapting systems, e.g., [25], but those works often lack intelligent fault-localization mechanisms.

To the best of our knowledge, our works are the first ones within the framework of consistency-based diagnosis which propose a general approach to the issue of dependent failures at the level of logical reasoning. The authors of [26] discuss the issue that multiple actions carried out by an agent may fail if the agent itself is in an abnormal state, and they regard the failure of those actions as dependent failures. The authors propose the use of causal rules which relate the health state of an agent to the abnormality of actions, and to use the health state to predict the failures of future actions. However, their approach is tailored to the diagnosis of agent plans and, even though it seems reasonable in their specific application, the general applicability is questionable. It does not solve the problems which we discussed in this paper.

The author of [27] proposes a method for reasoning with uncertainty in model-based diagnosis which enables stochastic dependencies among components; however, dependent failures are not addressed at the logical level.

Furthermore, the issue of dependent failures is similar to faults in the structure of the system which lead to hidden interactions, e.g., the bridge fault [15, 28, 29]. In both cases a single *cause of failure* may lead to multiple-fault diagnoses. However, in case of structural faults there is a single *point of failure*, whereas in presence of dependent faults multiple components have actually failed.

One advantage of our approach is the fact that, similar to [15], the complexity of the computation of diagnoses is not increased. I.e., it is possible to compute the minimal DEs only if required (e.g., when repair actions based on the minimal diagnoses have failed). Another strength is that our work is also able to recognize failure dependencies when there are multiple primarily failed components, whereas the authors of [28] and [15] assume that there is a single cause of failure.

The authors of [7] propose to compute *kernel diagnoses* instead of minimal diagnoses. However, we hold the view that in many applications multiple failures are only likely if they are dependent failures. Therefore, we explicitely model these dependencies by means of the failure dependency graph, which captures all possible failure dependencies. This graph is used to focus the reasoning.

11 Conclusion

We presented an approach which is able to automatically detect, locate and repair failures which occur in robot control systems at runtime. The proposed diagnosis system monitors the running system and detects violations of properties. In order to locate the failed components our approach employs the logical dependencies between the properties. We employ model-based diagnosis techniques for the computation of minimal diagnoses. However, model-based diagnosis approaches usually assume that components fail independently, and we have shown that dependent failures are quite common in robot control systems. The consequences

of dependent failures are that minimal diagnoses often do not contain all components which have failed, in particular they may not contain the independently failed component which has caused the dependent failures, and that the failed components often must be repaired in a specific order which corresponds to the causal order of the failures.

In order to tackle these problems we have proposed the concept of minimal diagnosis environments. Minimal DEs may contain further components which are not included in the minimal diagnoses, and they indicate the order in which components have failed. We have shown that the repair of a partly failed system is much more likely to succeed when it is based on minimal DEs rather than minimal diagnoses. Our approach requires that the model of the system is augmented with a failure dependency graph (FDG) which explicitely represents possible failure dependencies, and with a dependent failure description (*DFD*) which may contain fault models of dependently failed components. Moreover, we have proposed an algorithm which computes all minimal DEs of a system, and we have discussed the impact of dependent failures on repair strategies. Finally we have presented several challenging case studies in which we injected faults in a robot control system which led to dependent failures. The case studies clearly show that our approach is often able to correctly identify all failed components and the causal order of the failures, and that our diagnosis system can automatically repair the control system.

Besides its advantages, our approach also has a noteworthy drawback: it demands that the modeller has knowledge about the possible failure dependencies, and that he/she provides an additional model, the FDG. One might argue that the FDG could be automatically extracted from a system model. However, this would require that detailed models of the faulty behavior of components are available. In practice this can hardly be expected if the components are very complex, as it is the case for software services, because the programmers often are not able to provide sufficient informations. Moreover, creating such detailed behavior models is cumbersome and daunting for the involved programmers. Finally, extracting the knowledge from the source code is also unrealistic because of the program's size. By contrast, providing dependency information in the form of a FDG imposes little overhead. Anyway, the handling of dependent failures requires more knowledge about the system than it is the case for systems with only independent failures.

It should also be noted that the FDG is a non-local model, i.e., it is device-specific. This somehow contradicts the principle of compositional modeling which is a strength of the model-based diagnosis paradigm [5–7, 14, 28]. This principle means that the overall system model should be composed from the local models of the single components and the interconnections between the components. However, the diagnoses provided by classical model-based diagnosis do not indicate the failure cause-effect relationships, and we argue that the benefits of our approach outweigh the disadvantage of losing locality. Moreover, physical impossibility axioms [13], which are used in many practical model-based diagnosis approaches, may also violate the compositional modeling principle. Furthermore, notice that also in our approach the system description *SD* can rely on local behavior models only.

An interesting direction of future research is the automated extraction of failure dependency graphs from system descriptions. Such an approach might be possible in other domains where detailed and complete fault models are available.

References

1. Steinbauer G, Wotawa F (2005) Detecting and locating faults in the control software of autonomous mobile robots. In: Proceedings of the 19th international joint conf on artificial intelligence, Edinburgh, UK, pp 1742–1743
2. Peischl B, Weber J, Wotawa F(2006) Runtime fault detection and localization in component-oriented software systems. In: Proceedings of the 17th international workshop on principles of diagnosis (DX-06), Peñaranda de Duero, Spain, June 2006
3. Weber J, Wotawa F (2006) Using AI techniques for fault localization in component-oriented software systems. In: Proceedings of the 5th Mexican international conference on artificial intelligence (MICAI 2006), Apizaco, Mexico, November 2006
4. Weber J, Wotawa F (2007) AI-planning in a mobile autonomous robot with degraded software capabilities. In: International workshop on moving planning and scheduling systems into the real world, Providence, Rhode Island, USA, 2007. At ICAPS'07
5. Reiter R (1987) A theory of diagnosis from first principles. Artif Intell 32(1):57–95
6. de Kleer J, Williams BC (1987) Diagnosing multiple faults. Artif Intell 32(1):97–130
7. de Kleer J, Mackworth AK, Reiter R (1992) Characterizing diagnoses and systems. Artif Intell 56(2–3):197–222
8. Weber J, Wotawa F (2007) Diagnosing dependent failures in the hardware and software of mobile autonomous robots. In: Proceedings of the 20th international conference on industrial, engineering and other applications of applied intelligent systems (IEA/AIE 2007), Kioto, Japan, June 2007
9. Weber J, Wotawa F (2007) Diagnosing dependent failures—an extension of consistency-based diagnosis. In: 18th International workshop on principles of diagnosis (DX-07), Nashville, USA, 2007
10. Weber J, Wotawa F (2008) Dependent failures in consistency-based diagnosis. In: 18th European conference on artificial intelligence (ECAI 2008), pp 801–802, Patras, Greece
11. Hamscher W, Console L, de Kleer J (ed) (1992) Readings in model-based diagnosis. Morgan Kaufmann, San Mateo
12. Struss P, Dressler O (1989) Physical negation: Integrating fault models into the general diagnostic engine. In: Proceedings of the 11th international joint conf on artificial intelligence, pp 1318–1323
13. Friedrich G, Gottlob G, Nejdl W (1990) Physical impossibility instead of fault models. In: Proceedings of the national conference on artificial intelligence (AAAI). Also appears in Readings in Model-Based Diagnosis (Morgan Kaufmann, San Mateo 1992)

14. Davis R, Hamscher W (1988) Model-based reasoning Troubleshooting. In: Shrobe HE (ed) Exploring artificial intelligence. Morgan Kaufmann, San Mateo, pp 297–346, Chap. 8
15. Böttcher C (1995) No faults in structure? How to diagnose hidden interactions. In: IJCAI, pp 1728–1735
16. Steinbauer G, Mörth M, Wotawa F (2005) Real-time diagnosis and repair of faults of robot control software. In: Proceedings of the international RoboCup symposium, Osaka, Japan, 2005
17. de Kleer J (1990) Using crude probability estimates to guide diagnosis. Artif Intell 45:381–392
18. de Kleer J, Williams BC (1989) Diagnosis with behavioral modes. In: Proceedings of the 11th international joint conf. on artificial intelligence, pp 1324–1330
19. Minoux M (1988) LTUR: A simplified linear-time unit resolution algorithm for Horn formulae and computer implementation. Inf Process Lett 29:1–12
20. Grosclaude I (2004) Model-based monitoring of software components. In: Proceedings of the 16th European conference on artificial intelligence. IOS, Amsterdam, pp 1025–1026. Poster
21. Ardissono L, Console L, Goy A, Petrone G, Picardi C, Segnan M, Duprè DT (2005) Cooperative model-based diagnosis of web services. In: Proceedings of the 16th international workshop on principles of diagnosis, DX workshop series, pp 125–132, June 2005
22. Mayer W, Stumptner M (2003) Extending diagnosis to debug programs with exceptions. In: Proceedings of the 18th IEEE international conference on automated software engineering (ASE). Montreal, Quebec, Canada. IEEE Press, New York
23. Köb D, Wotawa F (2004) Introducing alias information into model-based debugging. In: 16th European conference on artificial intelligence (ECAI), Valencia, Spain, August 2004. IOS, Amsterdam, pp 833–837
24. Mikaelian T, Williams BC (2005) Diagnosing complex systems with software-extended behavior using constraint optimization. In: Proceedings of the 16th international workshop on principles of diagnosis, DX workshop series, pp 125–132
25. Garlan D, Schmerl B (2002) Model-based adaptation for self-healing systems. In: WOSS'02: Proceedings of the first workshop on self-healing systems. Assoc Comput Mach, New York, pp 27–32
26. Roos N, Witteveen C (2005) Diagnosis of plans and agents. In: Proceedings of the 4th international Central and Eastern European conference on multi-agent systems (CEEMAS)
27. Lucas PJF (2001) Bayesian model-based diagnosis. Int J Approx Reason 27(2):99–119
28. Davis R (1984) Diagnostic reasoning based on structure and behavior. Artif Intell 24:347–410
29. Preist C, Welham B (1990) Modelling bridge faults for diagnosis in electronic circuits. In: Proceedings of the first international workshop on principles of diagnosis, Stanford

Jörg Weber is a Ph.D. student at the Institute for Software Technology at Graz University of Technology (Austria). His general research interests include model-based diagnosis, Qualitative Reasoning, robotics, and the application of AI-planning to the deliberative control of autonomous robots.
His current research focuses on self-healing systems and, in particular, on runtime model-based diagnosis and repair/reconfiguration in mobile autonomous robots.
He received his M.Sc. degree in Computer Engineering (Telematik) from the Graz University of Technology.
He is a member of the Austrian Society for Artificial Intelligence.

Franz Wotawa received a M.Sc. in Computer Science (1994) and a Ph.D. in 1996 both from the Vienna University of Technology. He is currently a professor of software engineering, and head of the Institute for Software Technology (IST) at the Graz University of Technology. His research interests include model-based and qualitative reasoning, configuration, planning, theorem proving, intelligent agents, mobile robots, verification and validation, and software engineering. Currently, Franz Wotawa works on applying model-based diagnosis to software debugging as well as on testcase generation and repair. He has written more than 100 papers for journals, conferences, and workshops and has been member of the program committees for several workshops and conferences. He organized workshops and special issues on model-based reasoning for the journal AI Communications. He is in the editorial board of the Journal of Applied Logic (JAL), and a member of IEEE Computer Society, ACM, AAAI, the Austrian Computer Society (OCG), and the Austrian Society for Artificial Intelligence.