Runtime Fault Detection and Localization in Component-oriented Software Systems *

Bernhard Peischl and Joerg Weber and Franz Wotawa
Technische Universit¨at Graz
Institute for Software Technology
8010 Graz, Inffeldgasse 16b/2, Austria
Tel: +43 316 873 5723, Fax: +43 316 873 5706
{peischl,jweber,wotawa}@ist.tugraz.at

Abstract

In this paper we introduce a novel technique for run-time fault detection and localization in component-oriented software systems. Our novel approach allows to define arbitrary properties via rules at the component level. By monitoring the software system at run-time we can detect violations of these properties and, most notably, also localize possible causes for specific property violation(s). Relying on the model-based diagnosis paradigm, our fault localization technique is able to deal with intermittent fault symptoms and it allows for measurement selection. Finally, we discuss results obtained from our most recent case studies and relate our work to those of others.

1 Introduction

Several research areas are engaged in the improvement of software reliability during the development phase, for example research on testing, debugging, or formal verification techniques like model checking. Unfortunately, although substantial progress has been made in these fields, we have to accept the fact that faults in complex software systems are facts to be coped with, not problems to be solved [Patterson et al., 2002]. This perspective is supported by historical evidence and by numerous studies. Thus, it is highly desirable to augment complex software systems with autonomic fault localization capabilities, especially in systems which require high reliability.

The goal of our work is to detect and locate faults at run-time without any human intervention. Existing techniques like runtime verification aim at the detection of faults. However, it is necessary to locate faults in order to be able to automatically perform repair at runtime. Possible repair actions are, for example, the restart of software components or switching to redundant components.

In this paper we propose a technique for runtime fault detection and localization in component-oriented software systems. We define components as independent computational modules which have no shared memory and which communicate among each other by the use of events, which can contain arbitrary attributes. We suppose that the interactions are asynchronous. A component-oriented software system may be a single application which comprises loosely coupled processes (threads), or it may consist of multiple independent applications which communicate among themselves. The components may be split over a network. Typical implementations of the asynchronous event-based communication paradigm are, for example, CORBA, COM, JavaBeans, or even low-level communication methods like Unix message passing.

Moreover, we suppose that a certain event which is produced by a component can not be directly related to a specific incoming event, for example because incoming events may be internally queued. Furthermore, there may be connections which are not observable. Another assumption is that, as often the case in practice, no formalized knowledge about the application domain exists.

We require the runtime diagnosis to impose low run-time overhead in terms of computational power and memory consumption. Ideally, augmenting the software with fault detection and localization functionality necessitates no change to the software. We require the monitoring process to have no noticeable influence on the overall behavior of the system. Moreover, to avoid damage which could be caused by a faulty system, we have to achieve quick detection, localization, and repair. Another difficulty is the fact that the fault symptoms are often intermittent. One reason is that in runtime diagnosis the inputs to the system can not be kept constant while the diagnosis is performed, as the system continues operating. For example, a server application may receive new client requests during the fault localization process. In addition, software systems often operate in a physical environment which permanently changes, e.g. the control software of a mobile robot.

Our approach allows to introduce user-defined properties. The target system is continuously monitored by rules, i.e., pieces of software which detect property violations. The fact that the modeler can implement and integrate arbitrary rules provides sufficient flexibility to cope with today’s software complexity. In practice, properties and rules will often embody elementary insights into the software behavior rather than complete specifications. The reason is that, due to the

∗This research has been funded in part by the Austrian Science Fund (FWF) under grant P17963-N04. Authors are listed in alphabetical order.
complexity of software systems, often no formalized knowledge of the behavior exists and the informal specifications are coarse and incomplete.

In order to enable fault localization, complex dependences between properties can be defined. When a violation occurs, we locate the fault by employing the model-based diagnosis (MBD) paradigm [Reiter, 1987; de Kleer and Williams, 1987]. In terms of the classification introduced in [Brusoni et al., 1998], we propose a state-based diagnosis approach with temporal behavior abstraction. Furthermore, our model is able to deal with intermittent symptoms.

We evaluated our approach using the control software of a mobile autonomous robot as target system. The concrete models which we created for this system mainly aimed at the diagnosis of severe faults like software crashes and deadlocks.

Among the novel contributions of this paper is the monitoring of user-defined properties at the component level by integrating arbitrary rules. In particular, we employ relationships between properties for the localization of faults. Furthermore, we provide a formalization of the architecture of a component-based software system and of the property dependences, and we outline an algorithm for computing the logical model. We formally describe the diagnosis system and we present a runtime fault detection and localization algorithm which allows for measurement selection. Moreover, we give examples related to the control software of autonomous robots and discuss the results of case studies. Finally, we relate our work to those of others.

2 Introduction to the Model Framework

Figure 1: Architectural view on the software system of our example.

Figure 1 illustrates a fragment of a control system for an autonomous soccer robot as our running example. This architectural view comprises basically independent components which communicate by asynchronous events. The connections between the components depict data flows. The Vision component periodically produces events containing position measurements of perceived objects. The Odometry periodically sends odometry data to the WorldModel (WM). The WM uses probability-based methods for tracking object positions. For each event arriving at one of its inputs, it creates an updated world model containing estimated object positions. The Kicker component periodically creates events indicating whether or not the robot owns the ball. The Planner maintains a knowledge base (KB), which includes a qualitative representation of the world model, and chooses abstract actions based on this knowledge. The content of the KB is periodically sent to a monitoring application.

In [Steinbauer and Wotawa, 2005] an abstract behavior model of software components is proposed which is similar to the model in [Friedrich et al., 1999]. This model abstracts over concrete values in terms of functional dependences [Jackson, 1995]: If we assume a correctly working component and all inputs are correct, then the output(s) must be correct as well. In our example, the Planner component would be modelled as follows: \( \neg AB(\text{Planner}) \land ok(WS) \land ok(HB) \rightarrow ok(PS) \), where \( AB(c) \) denotes abnormality of component \( c \) and \( ok(e) \) states that a connection \( e \) is correct during a certain period of time. That is, the model abstracts from both the temporal constraints and the possibly complex values (logical contents) of events. Whether a connection is correct or not is determined by observers. An observer comprises rules, which are pieces of software which monitor certain parts of the software system. For example, the observers for \( ok(OM) \), \( ok(MD) \), and \( ok(PS) \), would contain rules which continuously check if events on these connections are produced periodically.

While the model in [Steinbauer and Wotawa, 2005] proved applicable in various settings, we argue that in many cases this model is too abstract to express software behavior. As a matter of fact, a component’s complex behavior cannot be captured by simple dependences.

First, a separation of temporal constraints and constraints related to the values of events is highly desirable. For example, in Figure 1, the Planner is supposed to produce events on the connection PS periodically, regardless of the inputs to this component. Thus this constraint does not depend on any input, and from its violation we can directly infer that the Planner has failed. However, the value of the PS events is directly influenced by the Planner’s inputs.

Second, as events may contain complex values, fine-grained dependences are necessary for capturing the real behavior. For example, some parts of the knowledge base, whose content is transmitted over the PS connection, depend on the world model, i.e. on the WS connection, while other parts depend on the HB connection.

Our new model addresses these issues by assigning a set of properties, i.e. constraints, to components and connections. In the logical model, these properties are represented by property constants. We use the proposition \( \text{ok}(x, pr, s) \) which states that the property \( pr \) holds for \( x \) during a certain period of time, where \( x \) is either a component or a connection. While the system is continuously monitored by the rules, the diagnosis itself is based on (multiple) discrete snapshots of the system. The snapshots are obtained by polling the states of the rules (violated or not violated) at discrete time points. Each observation belongs to a certain snapshot, and we use the variable \( s \) as a placeholder for a specific snapshot. The diagnosis accounts for the observations of all snapshots. This approach to MBD is called multiple-snapshot diagnosis or state-based diagnosis [Brusoni et al., 1998].

An example for a component-related property is \( pr_{np} \), expressing that the number of processes (threads) of a correctly
working component $c$ must exceed a certain threshold. In our running example, $pr_{pe}$ denotes that events must occur periodically on a connection, and $pr_{cons,c}$ is used to denote that the value of events on a certain connection must not contradict the events on connection $c$.

The observer for $ok(WS, pr_{cons,OM}, s)$ checks if the computed world models on connection WS correspond to the object position measurements on connection OM. Ideally, such an observer would embody a complete specification of the tracking algorithm used in the WM component. In practice, however, often only incomplete and coarse specifications of the complex software behavior are available. Therefore, the observers rely on simple insights which require little expert knowledge. The rules of the observer for $ok(WS, pr_{cons,OM}, s)$ could check if all environment objects which are perceived by the Vision are also part of the computed world models, disregarding the actual positions of the objects (note that the set of perceived objects often changes in a dynamic environment, and those objects which are no longer perceived will be tracked by the WM for a while and finally discarded). Our experience has shown that such abstractions often suffice to detect and locate severe faults like software crashes or deadlocks.

Using such properties, the dependences between the inputs and outputs of components can be refined, as the logical model in Figure 3 shows. Figure 2 depicts the properties which we assign to the connections, and Figure 4 shows the dependences between properties on the input and output connection of the WM and the Planner.

The model captures, for example, that the WM must generate events periodically, provided that the temporal constraints on the incoming connections hold. Furthermore, the value of the events on connection WS must be consistent with the OM connection, provided that the events on OM occur periodically.

To illustrate our basic approach we outline a simple scenario by locating the cause for observed malfunctioning. We assume a fault in the WM causing the world state WS and, as a consequence, the planner state PS to become inconsistent with the object position measurements OM. As a result, the observer for $ok(WS, pr_{cons,OM}, s)$ detects a violation, i.e. $\neg ok(WS, pr_{cons,OM}, s_0)$ is an observation for snapshot 0. All other observers are initially disabled, i.e. they do not provide any observations.

Based on this observation, we can compute diagnosis candidates by employing the MBD [Reiter, 1987; de Kleer and Williams, 1987] approach for this observation snapshot. By computing all (subset minimal) diagnoses, we obtain three single-fault diagnosis candidates, namely $\{AB(Vision)\}$, $\{AB(OM)\}$, and $\{AB(Plan)\}$. Note that, using the coarse-grained model in [Steinbauer and Wotawa, 2005], the Odometry and the Kicker would be candidates, too.

After activating observers for the output connections of these candidates, we obtain the second observation snapshot $ok(OM, pr_{pe}, s_1)$, $ok(WS, pr_{pe}, s_1)$, $\neg ok(WS, pr_{cons,OM}, s_1)$, $ok(PS, pr_{pe}, s_1)$, $\neg ok(PS, pr_{cons,OM}, s_1)$, and $ok(PS, pr_{cons,HB}, s_1)$. This leads to the single diagnosis $\{AB(WM)\}$.

Let us consider a second scenario related to Figure 2. It demonstrates that our model framework allows refinements which may lead to the correct identification of multiple-fault diagnoses in situations in which a less fine-grained model would find solely single-fault diagnoses.

We assume that monitoring the connections OM and MD is either impossible or unrealistic due to high costs. Suppose that no events on connection WS occur, thus $\neg ok(WS, pr_{pe}, s_0)$ is observed. Given the model in Fig. 3, we obtain 3 single-fault diagnoses: $\{AB(Vision)\}$, $\{AB(OM)\}$, and $\{AB(WM)\}$. However, the WM generates an output event for each event on one of its incoming connections. Therefore, if only one of the components Vision or Odometry were faulty, the number of events on the connection WS would still be larger than 0. As a consequence, we can conclude that either the WM or both the Vision and the Odometry have failed.

We gain a better result by refining the model as shown in Figure 5. The sentences in Figure 5 extend the model in Figure 3. The new property $pr_{co}$ holds only if at least one event occurs on a connection during a certain time period. A new sentence is added to the model of the WM component. It states that, if the WM works correctly and the property $pr_{co}$ holds for at least one input connection, then it must hold for the connection WS as well. Note that we use a kind of dependence for $pr_{co}$ that is different from what we have seen so far. We will call this a partial dependence in Section 3.

Now the observers detect two property violations: $\neg ok(WS, pr_{pe}, s_0)$ and $\neg ok(WS, pr_{co}, s_0)$. We obtain a single-fault diagnosis $\{AB(WM)\}$ and a single dual-fault diagnosis $\{AB(Vision), AB(ODometry)\}$, which ob-
viously resembles the human kind of reasoning.

Figure 4: Graphical representation of dependences in Fig. 3 for two example components.

\[\neg AB(Vision) \rightarrow ok(OM, pr_{eo}, s)\]
\[\neg AB(Odometry) \rightarrow ok(MD, pr_{eo}, s)\]
\[\neg AB(WM) \land (ok(OM, pr_{eo}, s) \lor ok(MD, pr_{eo}, s))\]
\[\rightarrow ok(WS, pr_{eo}, s)\]

Figure 5: Extension of the model in Fig. 3.

3 Formalizing the Model Framework

In Definition 3.1 we introduce a model which captures the architecture of a component-oriented software system and the dependences between properties.

**Definition 3.1 (SAM)** An software architecture model (SAM) is a tuple \((COMP, CONN, \Phi, \varphi, out, in_p, in_t)\) with:
- a set of components \(COMP\)
- a set of connections \(CONN\)
- a (finite) set of properties \(\Phi\)
- a function \(\varphi : COMP \cup CONN \rightarrow 2^\Phi\), assigning properties to a given component or connection.
- a function \(out : COMP \rightarrow 2^{CONN}\), returning the output connections for a given component.
- the (partial) functions \(in_p : COMP \times CONN \times \Phi \rightarrow 2^{CONN\times\Phi}\) which express the functional dependences between the inputs and outputs of a given component \(c\). For all output connections \(e \in out(c)\) and for each property \(pr \in \varphi(e)\), they return a set of tuples \((e', pr')\), where \(e'\) is an input connection of \(c\) and \(pr' \in \varphi(e')\) a property assigned to \(e'\).

This definition allows to specify a set of properties \(\Phi\) for a specific software system. We introduce a function \(\varphi\) in order to assign properties to components and connections.

The functions \(in_p\) and \(in_t\) formalize the functional dependences between properties of the inputs and of the outputs. For each property \(pr\) of an output connection, they return a set of input properties \(PR\) on which \(pr\) depends. Function \(in_t\) expresses total dependences: if a component is correct and all properties in \(PR\) hold, then \(pr\) must hold as well. By contrast, \(in_p\) defines partial dependences: if a component is correct and at least one property in \(PR\) holds, then \(pr\) must hold, too. In our example, the dependence of \((WS, pr_{eo})\) on \((OM, pr_{eo})\) and \((MD, pr_{eo})\) is partial (see Fig. 5). All other dependences in this example are total.

Note that [Friedrich et al., 1999] and [Steinbauer and Wotawa, 2005] use only one kind of functional dependence, which is equivalent to what we call a total dependence herein.

For example, those part of the SAM which relates to the WM component and its output connection WS are defined as follows (Fig. 3 and 5):

\[\varphi(WM) = \{pr_{np}\}, \varphi(WS) = \{pr_{cons}, pr_{cons,OM}, pr_{eo}\}\]
\[out(WM) = \{WS\}\]
\[in_p(WM, WS, pr_{eo}) = \{(OM, \{pr_{cons}\}), (MD, \{pr_{eo}\})\},\]
\[in_t(WM, WS, pr_{cons,OM}) = \{(OM, \{pr_{cons}\}), (MD, \{pr_{eo}\})\}\]

The logical model is computed by Algorithm 1. Based on a SAM, it generates the logical system description \(SD\). In line (3), we create those sentences which relate to component properties. In line (4), a logical representation of the dependences between properties is computed. It is distinguished between total and partial dependences.

Note that the universal quantification implicitly applies to variable \(s\). It denotes a discrete snapshot of the system behavior. Each observation \(\neg ok(x, pr, s_i)\) relates to a certain snapshot \(s_i\), where \(i\) is the snapshot index. A diagnosis is a solution for all snapshots. The temporal ordering of the different snapshots is not taken into account.

It is also important that, supposed that the number of snapshots is finite, the logical model which is computed by this algorithm can be easily transformed to propositional Horn clauses and thus the model is amenable to efficient logical reasoning.

4 Runtime Monitoring and Fault Localization

The runtime diagnosis system consists of two modules, the diagnosis module (DM) and the observation module (OM). These modules are executed concurrently. While the DM performs runtime fault detection and localization at the logical level, the OM continuously monitors the software system and provides the abstract observations which are used by the DM. Thus, the OM can be regarded as an abstraction layer between the architecture model, as presented in Section 3, and the running software.

Let us consider the OM first. It basically consists of observers. Each observer comprises a set of rules which specify the desired behavior of a certain part of the software system. A rule is a piece of software which continuously monitors that part. The execution of the rules is concurrent and unsynchronized, and we do not impose any restrictions on the implementation of a rule and its complexity. Furthermore, while
Algorithm 1: The algorithm for computing the logical model.

Input: The set of enabled observers and a constant denoting the current snapshot.

Output: The system description SD.

\begin{algorithmic}
\State SD := \{\}\; 
\State For all e ∈ COMP:\ 
\State (3) For all pr ∈ ϕ(e): add \neg \text{AB}(c) → ok(c,pr,s) to SD.
\State (4) For all e ∈ out(e), for all pr ∈ ϕ(e): add \neg \text{AB}(c) \land \bigwedge_{(e',pr') \in \text{in}_{\text{t}}(c,e,pr)} \text{ok}(e',pr',s) \rightarrow \text{ok}(e,pr,s)
\State and \neg \text{AB}(c) \land \bigwedge_{(e',pr') \in \text{in}_{\text{f}}(c,e,pr)} \text{ok}(e',pr',s) \rightarrow \text{ok}(e,pr,s)
\State to SD.
\State (5) Return SD.
\end{algorithmic}

a property is assigned to a single component or connection, a rule may monitor multiple communication links in the target system in order to detect wrong sequences of events. In our example, the rules for \text{ok}(\text{WS},pr_{cons},\Omega)\text{OM},s) take the events on the connections WS and OM into account.

When a rule detects a violation of its specification, it switches from state not violated to the state violated. To each observer a set of atomic sentences is assigned which represent the logical observations.

Furthermore, an observer may be enabled or disabled. The rules of a disabled observer are inactive, and the observer does not provide any observations. Disabled observers may be enabled in the course of the fault localization. Note that it is often desired to initially disable those observers which otherwise would cause unnecessary runtime overhead.

Definition 4.1 (Observation Module) The OM is a tuple (OS,OS\text{e}), where OS is the set of all available observers and OS\text{e} \subseteq OS the set of those observers which are currently enabled.

Definition 4.2 (Observer) An observer \text{os} ∈ OS is a tuple \text{pr}(\Omega,\text{om}) with:

1. a set of rules \text{R}.
2. a set of atomic sentences \text{Ω}.

Algorithm 2: The algorithm for computing the set of observations.

Input: The set of enabled observers and a constant denoting the current snapshot.

Output: The set OBS which comprises ground denoting the current snapshot.

\begin{algorithm}
\State OBS := \{\}\; 
\State For all os ∈ OS\text{e}, os = (R,Ω): 
\State (3) If os \in v(OS\text{e}): add \bigwedge_{\omega \in \text{OBS}} \neg \omega \rightarrow \text{OBS}
\State (4) else: add \bigwedge_{\omega \in \text{OBS}} \omega \rightarrow \text{OBS}.
\State (5) For all atoms \alpha ∈ OBS: substitute \text{s}_i for the variable \text{s}.
\State (6) Return OBS.
\end{algorithm}

Algorithm 3 presents the algorithm which is executed by the diagnosis module DM. The inputs to the algorithm are the logical system description SD, which is returned by the computeModel algorithm (Alg. 1), and an observation module \text{OM} = (OS,OS_\text{e}). In contrast to the work in [Steinbauer and Wotawa, 2005], this algorithm is able to gather additional observations by integrating runtime measurement selection.

The algorithm periodically determines whether a misbehavior is detected by an observer. In this case, it waits for a certain period of time (line 6). This gives the observers the opportunity to detect additional symptoms, as it may take some time after faults manifest themselves in the observed system behavior. Thereafter, the diagnoses are computed (line 10) using Reiter’s Hitting Set algorithm [Reiter, 1987].

Note that the violated rules are reset to not violated after computing the logical observations (line 9). Therefore, an observer which detects a misbehavior in snapshot \text{s}_j may report a correct behavior in \text{s}_{j+1}. This is necessary for the localization of multiple faults in the presence of intermittent symptoms.

When we find several diagnoses (lines 11 and 12), it is desirable to enable additional observers in OS \setminus OS_{\text{e}}. We assume the function ms(\text{SD,OBS},OS,OS_{\text{e}}) to perform a measurement selection, i.e. it returns a set of observers OS_{\text{e}} (OS_{\text{e}} ⊆ OS \setminus OS_{\text{e}}) whose observations could lead to a refinement of the diagnoses. We do not describe the function ms in this paper. In [de Kleer and Williams, 1987] a strategy based on Shannon entropy to determine the optimal next measurement is discussed. Note that the returned set may be empty, even if no unique diagnosis is derivable.

The fault localization is finished when either a unique diagnosis is found or the diagnoses can not be further refined by enabling additional observers (line 11).

5 Case Studies and Discussion

We implemented the proposed diagnosis system and conducted a series of experiments using the control software of a mobile autonomous soccer robot. We applied a propositional Horn clause theorem prover for consistency checks in the diagnosis engine [Minoux, 1988]. The implemented measurement selection process may enable multiple observers at the same time in order to reduce the time required for fault localization.

The components of the control system are executed in
Algorithm 3: The runtime diagnosis algorithm.
Input: The logical system description and the observation module.
PERFORMRUNDIENGESSION($SD, OM$)
(1) Do forever:
(2) Query the observers, i.e. compute the set $v(OS_e)$.
(3) If $v(OS_e) \neq \emptyset$:
(4) Set $i := 0$, $OBS := \emptyset$, $finished := false$, where $i$ is the snapshot index.
(5) While not finished:
(6) Wait for the symptom collection period $\delta_c$.
(7) Recompute $v(OS_e)$.
(8) $OBS := OBS \cup OBS_e$, where $OBS_e := computeOBS(OS_e, s_i)$
(9) Reset all rules to not violated.
(10) Compute $D$: $D := \{\Delta | \Delta$ is a minimal diagnosis of (SD, COMP, OBS)$\}$.
(11) If $|D| = 1$ or the set $OS_s := ms(SD, OBS, OS, OS_s)$ is empty: start repair, set $finished := true$.
(12) Otherwise: set $i := i + 1$, enable observers in $OS_s$, and set $OS_e := OS_e \cup OS_e$.

We simulated software failures by killing single processes in 10 different applications and by injecting deadlocks in these applications. We investigated if the faults can be detected and located in case the outputs of these components are observed. In 19 out of 20 experiments, the fault was detected and located within less than 3 seconds. In only one case it was not possible to detect the fault because the specification of an important connection would have required information about the physical environment which was not available. Note that we set the symptom collection period $\delta_c$ to 1 second (see Alg. 3, line 6), and the fault localization incorporated no more than 2 observation snapshots.

Due to the small number of components and connections, the computation of the diagnoses required only a few milliseconds. Furthermore, the overhead (in terms of CPU load and memory usage) caused by the runtime monitoring was negligible, in particular because calls to the diagnosis engine are only necessary after an observer has detected a misbehavior.

Furthermore, we conducted 6 non-trivial case studies in order to investigate more complex scenarios. We injected deadlocks in different applications. We assumed that significant connections are either unobservable or should be observed only on demand, i.e. in course of the fault localization, because otherwise the runtime overhead would be unacceptable. In 4 scenarios we injected single faults, while in the other cases 2 faults occurred in different components almost at the same time. Moreover, in 2 scenarios the symptoms were intermittent and disappeared during the fault localization.

We gained several insights from our experiments. In general, state-based diagnosis appears to be an appropriate approach for fault localization in a robot control system as a particular example for component-oriented software. We were able to identify simple patterns in the interaction among the components, and by using rules which embody such patterns it was possible to create appropriate models which abstract from the dynamic software behavior. Furthermore, the approach proved to be feasible in practice since the overhead caused by the runtime monitoring is low.

An important issue is how to find the properties for a specific application. Our work aims at software systems comprising large and complex components. At present, for such systems it is rarely the case that formal specifications are available. Thus, it will often be necessary to manually derive the properties from informal (textual and graphical) specifications, which are often coarse and incomplete.

It would be desirable to automatically extract the properties from the source code of the software system, for example by relying on assertions (Design by Contract, [Meyer, 1997]). Unfortunately, in general this is not possible due to several reasons. First, only a part of the source code of a software system may be available, especially in complex systems which often integrate third party frameworks and libraries. Second, we cannot expect that the automated extraction of properties is computationally feasible for complex systems. The granularity of properties at the component level is quite different from that of assertions at the source code level, as properties relate to the overall behavior of a component whereas assertions are assigned to functions and classes in the source code and thus define local conditions. Therefore, in order to derive a single property automatically it would, in general, be necessary to take the entire source code (including all assertions) into account, which is computationally infeasible for large systems.

A main problem is the fact that simple rules are often too
coarse to express the software behavior. Such rules may detect faults only in certain situations. Therefore, it may happen that faults are either not detected or that they are detected too late, which could cause damage due to the misbehavior of the software system. The usage of simple rules also has the effect that more connections must be permanently observed than it would be the case if more complex rules were used. For example, in the control system we used in our experiments we had to observe more than half of the connections permanently in order to be able to detect severe faults like deadlocks in most of the components.

6 Related Research

There is little work which deals with model-based runtime diagnosis of software systems. In [Grosclaude, 2004] an approach for model-based monitoring of component-based software systems is described. The external behavior of components is expressed by Petri nets. In contrast to our work, the fault detection relies on the alarm-raising capabilities of the components themselves and on temporal constraints.

In the area of fault localization in Web Services, the authors of [Ardissono et al., 2005] propose a modelling approach which is similar to ours. Both approaches use grey-box models of components, i.e. the dependences between the inputs and outputs of components are modelled. However, their work assumes that each message (event) on a component output can be directly related to a certain input event, i.e. each output is a response which can be related to a specific incoming request. As we can not make this assumption, we abstract over a series of events within a certain period of time.

Another approach to model the behavior of software is presented in [Mikaelian and Williams, 2005]. In order to deal with the complexity of software, the authors propose to use probabilistic, hierarchical, constraint-based automata (PHCA). However, their work addresses software which is embedded in hardware systems, and they model the software in order to detect faults in the hardware. The authors of [Mikaelian and Williams, 2005] do not detect software bugs.

In the field of autonomic computing, there are model-based approaches which aim at the creation of self-healing and self-adaptive systems. The authors of [Garlan and Schmerl, 2002] propose to maintain architecture models at runtime for problem diagnosis and repair. Their architecture models comprise components and connectors. Their notion of a component resembles our definition. Similar to our work, they assign properties to components and connectors. The constraints over the properties are defined in a first-order language. However, their work does not employ fault localization mechanisms.

Pinpoint [Chen et al., 2002] is a framework for root-cause analysis in large distributed component applications (e.g. e-commerce systems). Pinpoint monitors client requests, uses traffic sniffing and middleware instrumentation to detect failed requests, and then applies data mining techniques to determine which components are likely to be faulty. The advantage of their approach is that is does not rely on static dependency models. No knowledge of the application components is required.

The author of [Auguston, 1998] suggests an approach to assertion checking, debugging, and profiling by building a behavioral model in terms of a number of events (so called event traces). Moreover, the author proposes a language to describe computations over event traces and states that algorithmic debugging [Shapiro, 1983] can be considered as an example of a debugging strategy based on a specific assertion language (e.g. assertions about procedure call outcomes). Moreover, the authors of [Console et al., 1993] discuss the relationship between algorithmic debugging and MBD.

In contrast to the work presented herein, research in the area of model-based software debugging deals with verification and particularly fault localization [Mayer and Stumptner, 2003; Köb and Wotawa, 2004] at compile time. Since limitations on computational and memory resources are less stringent than in runtime diagnosis, most of this research deals with fault localization at the object, statement or expression level, whereas our model focuses on capturing component-level behavior.

Design by Contract [Meyer, 1997] is a lightweight formal technique for runtime detection of specification violations. The trace assertions approach [Brökens and Möller, 2002b; 2002a] extends the Design by Contract approach by specifying the desired behavior of a program in terms of CSP-like processes. This allows for specifying valid events in a systematic fashion also incorporating abstraction techniques. Similar to our approach these so-called trace assertions are checked at runtime. However, the work presented in [Brökens and Möller, 2002b; 2002a] focuses on runtime error detection in Java programs and on specification techniques for traces. Our work fits in the same context, however, we focus on fault detection as well as localization in particular in autonomous software systems.

The authors of [Steinbauer and Wotawa, 2005] discuss the repair of component-oriented software systems at runtime. The repair is basically done by restarting failed components.

7 Conclusion and Future Research

This paper presents a model-based diagnosis approach for the detection and localization of faults in component-oriented software systems at runtime. Our model allows to introduce arbitrary properties and to assign them to components and connections. The fault detection is performed by rules, i.e. pieces of software which continuously monitor the software system in order to detect property violations. The fault localization utilizes dependences between properties. We formalize the architecture of a component-oriented software system and the dependences between properties. We employ two different kinds of dependences, total dependences and partial dependences.

Moreover, we provide algorithms for the generation of the logical model and for the runtime diagnosis. The runtime fault localization integrates measurement selection by enabling additional observers at runtime.

Finally, we discuss case studies which demonstrate that our approach is frequently able to quickly detect and locate faults. We were able to create appropriate models which abstract from the dynamic behavior by relying on simple rules which embody elementary insights into the software system.
The main problem we identified is the fact that simple rules, in contrast to more complex specifications, often detect faults only in certain situations. As a consequence, it may happen that faults are either not detected or they are detected too late.

We plan to evaluate our approach in other application domains as well. Another open issue is if our approach can be adapted to a distributed diagnostic engine, which would be useful for software systems which are distributed over a network. Moreover, our future research will deal with autonomous repair of software systems at runtime.

References


