Detecting and locating faults in the control software of autonomous mobile robots *

Gerald Steinbauer and Franz Wotawa
Technische Universität Graz
Institute for Software Technology
8010 Graz, Inffeldgasse 16b/2, Austria
Tel: +43 316 873 5723, Fax: +43 316 873 5706
{steinbauer,wotawa}@ist.tugraz.at

Abstract

Research on diagnosis has a long history in artificial intelligence which includes work dealing with diagnosis and repair of autonomous systems like space probes and software debugging. In this paper we extend previous work by introducing modeling principles for software architectures that allow to detect and locate failures of the robot’s control software. Failure detection is based on observers, i.e., software that monitors the behavior of the control software. Localization is based on a model of the architecture and uses model-based diagnosis for computing the failure causes. Moreover, we introduce an algorithm for repairing the software and discuss first experimental results of our implementation.

1 Introduction

Control software of autonomous and mobile robots is characterized by its fairly high complexity which is in conflict with runtime requirements like stability and flexibility. Complexity is caused by the software components implementing the basic functionality like planning, computing world models, sensor and actuator interfaces, and their connections. Because of the high complexity but also the instability of hardware components and connections, and the underlying operating system, complete stability of such a platform is very unlikely. This problem description does not only hold for mobile robots but all systems comprising software and hardware which interact with the real world. But when we want to build a robot that is truly autonomous it has to deal with failures during its runtime without degrading the desired behavior or even worse failing to fulfill its mission. Hence, a diagnosis system on top of the control system which does monitoring the current behavior, locating the cause of a detected failure, and taking the appropriate actions is a necessity.

There are several requirements for a diagnosis system in the domain of mobile robots. First, ideally the diagnosis system should not cause any changes of the control systems. If changes are necessary, they should be as small as possible. Furthermore, the diagnosis system must not affect the behavior of the control system. This requirement is very important in order to keep the effort for introducing a diagnosis system as small as possible. Second, the diagnosis system should not reduce the overall available computational power because this might decrease the robot’s functionality, e.g., its ability to react to a given event in a certain amount of time. Third, in cases where the diagnosis system itself does fail, there should be (almost) no effects on the control system. Finally, the memory requirements of the diagnosis system should be as small as possible. Otherwise, the diagnosis system has a too large effect on the system performance. For critical applications it is desirable that the integration of a diagnosis system is part of the initial design.

In order to fulfill the above requirements we introduce a model-based solution. This includes a model of software components and their relationships that are specified in the software architecture of the robot’s control system. This model is then used to derive root causes of a detected failure. Root causes itself are software components. The failure detection is based on observations. For this purpose we use the concept of observers, i.e., software programs that monitor system activities like the number of active processes of a software component. If the monitored value exceeds its pre-specified boundaries, the observer raises a conflict which causes the diagnosis engine to compute the root causes. Once the root cause has been identified, the diagnosis system takes appropriate actions in order to retain the system’s correct behavior. Possible actions are killing and re-generating processes that cause the failure. The fault detection, localization, and correction procedures are all based on declarative models of the control software.

In this paper, we present the modeling paradigms, the observers, and the algorithm for retaining the correct state. Moreover, we present the results of a case study which had been realized by modeling the control software architecture of our mobile robot. In the used test-cases typical failures, like software components that become inactive because of deadlocks, are represented. The case study shows that the overall system performance is not degraded and that the diagnosis system always retains the desired state.

*This research has been funded in part by the Austrian Science Fund (FWF) under grant P17963-N04. Authors are listed in alphabetical order.
2 Modeling software architectures

Software architectures provide a general view on software. Software architectures comprise software components and their connections. Components represent a collection of classes which implement a certain behavior. The connections between components represent dependency relations like client-server relationships and data flow. For example a robot’s architecture might comprise components for image processing, motion control, planning actions, and others. During the execution of a program the components might spawn processes and interact using method calls, or other means of communication, like events. Figure 1 depicts parts of the software architecture of our mobile robot.

![Diagram of software architecture](image)

Figure 1: Dependencies between software and hardware modules

The following formalization of the structural properties of the software architecture considers the software components, their connections in terms of identifiers representing events or procedure calls, and a classification of dependency relations which are used to repair the software during runtime. We distinguish two different dependency relations between components: weak and strong dependencies. Two component are weakly dependent if killing one component at runtime does not require the other component to be killed and to be restarted in order to repair the overall system. Otherwise, the relationship is a strong dependency.

**Definition 2.1 (SAM)** A software architecture model (SAM) is a tuple \((CO, C, out, in, WDC, SDC)\) with:

- a set of software components \(CO\)
- a set of connections \(C\)
- A function \(out : CO \mapsto 2^C\) returning the output connections for a given component
- A function \(in : CO \times C \mapsto 2^C\) returning the input connections for a given component and output connection. This function only returns those inputs that influence the value of the specified output.
- a set of weak dependencies \(WDC \subseteq 2^{CO \times CO}\)
- a set of strong dependencies \(SDC \subseteq 2^{CO \times CO}\)

We represent all weak and strong dependencies as tuple \((x_1, x_2)\) with \(x_1, x_2 \in CO\). The direction of the connection is from \(x_1\) to \(x_2\).

Hence, the SAM representing the software architecture of Fig. 1 is:

\[
\begin{align*}
&\{\{\text{LASER}, \text{CAN}, \text{OT}, \text{GL}, \text{MO}, \text{SO}, \text{KI}, \text{WM}, \text{BE}, \text{PL}\} \\
&\{\text{ObjectMeasurement}, \text{WorldState}, \ldots\} \\
&\{\text{out}(\text{MO}) = \{\text{MotionDelta}, \ldots\}, \\
&\{\text{in}(\text{MO}, \text{MotionDelta}) = \{\text{CAN}_1, \ldots\}, \\
&\{(\text{WM}, \text{OT}), \ldots\}, \{(\text{LASER}, \text{GL}), \ldots\}\}
\end{align*}
\]

where \(OT\) is the object tracker, \(GL\) the goal locator, \(MO\) motion, \(SO\) sonar, \(KI\) kicker, \(WM\) world model, \(BE\) behavior engine, and \(PL\) planner.

The concrete behavior of the software at runtime is determined by the implemented behavior of its software components. A formalization of the concrete behavior requires the transformation of the whole program which is not only a very difficult task but leads to models that can hardly be used for diagnosis at runtime where resources for diagnosis are limited. Hence, an abstraction of the concrete behavior is necessary. The idea behind the abstract behavior model of software components is similar to models which are based on dependencies like the one described by [Friedrich et al., 1999]. If all inputs to the model are correct, a software component should produce a correct output. This conversion has to be performed for all components and their output connections.

The algorithm for performing this conversion is as follows where the predicate \(\text{AB}\) stands for abnormal, and \(ok\) indicates a correct event or method call:

\[
\text{computeModel}((CO, C, out, in, WDC, SDC))
\]

**Input:** The SAM.

**Output:** A set of horn clauses

1. Let \(M\) be the empty set.
2. For all \(x \in CO\):
   (a) Add \(\neg \text{AB}(x) \rightarrow \text{ok}.\text{num}.\text{processes}(x)\) to \(M\).
   (b) For all \(e \in \text{out}(x)\) add \(\neg \text{AB}(x) \land \bigwedge_{e' \in \text{in}(x,e)} \text{ok}(e') \rightarrow \text{ok}(e)\) to \(M\).
3. Return \(M\).

Line 2(a) introduces a rule which says that a correct component spawns the correct number of processes at runtime. Because this parameter of a software component can be easily checked via operating systems call, we incorporate this knowledge in our model.

For example, the rule that represents the abstract behavior of the \(OT\) (Object Tracker) component is:

\[
\neg \text{AB}(\text{OT}) \land \text{ok}(\text{Firewire}) \rightarrow \text{ok}(\text{ObjectMeasurement})
\]

The size of the model in terms of number of literals depends on the number of components, the maximum fan-in, and the maximum fan-out of the components. The fan-in and the fan-out are both bound by the cardinality of the connections.
Theorem 2.1 The number of literals of the model returned by calling \textit{computeModel}((\textit{CO}, \textit{C}, \textit{out}, \textit{in}, \textit{WDC}, \textit{SDC})) is $O(|\textit{CO}| \cdot |\textit{C}|^2)$.

If the maximum fan-in and the maximum fan-out are much smaller than the number of components, the number of literals is of order $O(|\textit{CO}|)$ which is almost always the case for practical applications.

In order to locate root causes, i.e., the components of the software architecture which cause a detected misbehavior, we have to introduce a notation of observations at the same conceptual level. The easiest way of doing this (which has also been done by [Friedrich et al., 1999]) is to use the same \textit{ok} predicate for the purpose. If for example we detect a misbehavior at \textit{ObjectMeasurement}, we could represent this by the literal $\neg \textit{ok}(\textit{ObjectMeasurement})$. The drawback of this representation is the impossibility of distinguishing observations and computed values. Hence, it would be better to introduce a distinguished predicate \textit{correct} for observations.

Definition 2.2 (Observation) Given a SAM (\textit{CO}, \textit{C}, \textit{CS}). Either \textit{correct}(x) or $\neg \textit{correct}(x)$ are observations for a connection \textit{x}. \textit{correct}(x) is true whenever the observed connection shows the correct behavior. If we observe a failure for \textit{x}, the observation has to be $\neg \textit{correct}(x)$.

The final step for generating the model is to add rules for coupling observations to models generated by \textit{computeModel}. The following algorithm provides this information.


\textbf{Input:} A set of horn clauses representing the interface between the architecture model and observations.

1. Let \textit{M} be the empty set.
2. For all $e \in \textit{C}$ add \textbf{correct}(e) $\rightarrow$ \textbf{ok}(e) and $\neg \textbf{correct}(e) \rightarrow \neg \textbf{ok}(e)$ to the model \textit{M}.
3. For all $x \in \textit{CO}$ add \[
\bigwedge_{e' \in \text{out}(x)} \textbf{correct}(e') \rightarrow \neg \textbf{AB}(x)
\]
to the model \textit{M}.
4. For all $x \in \textit{CO}$ add \textbf{correct}(\textit{processes})(x) $\rightarrow$ \textbf{ok}\_\textit{num}\_\textit{processes}(x)
to the model \textit{M}.
5. Return \textit{M}.

Line 2 of \textit{computeOBSModel} provides the interface between the observations and the derived values. Line 3 captures a case where everything that is computed by a software component is known to be correct. In this case it is very likely that the component itself is correct which is represented by line 3. In line 4 we provide an interface to the number of processes counter because every component spawns a fixed number of processes. Because the number of processes for each component is only a necessary condition for the correctness of a component it is not correct when saying that the correct number of processes implies the correctness of the component. Therefore, we do not add such a rule to our model.

For our example \textit{computeOBSModel} would return the following rules for \textit{OT} and \textit{ObjectMeasurement}:

\[
\text{correct}(\textit{ObjectMeasurement}) \rightarrow \text{ok}(\textit{ObjectMeasurement})
\]

\[
\neg \text{correct}(\textit{ObjectMeasurement}) \rightarrow \neg \text{ok}(\textit{ObjectMeasurement})
\]

\[
\text{correct}(\textit{ObjectMeasurement}) \rightarrow \neg \text{AB}(\textit{OT})
\]

Theorem 2.2 The number of literals of the interface model returned by calling \textit{computeOBSModel}((\textit{CO}, \textit{C}, \textit{out}, \textit{in}, \textit{WDC}, \textit{SDC})) is $O(|\textit{C}| \cdot |\textit{CO}|)$.

3 Monitoring events, method calls, and processes

Coupling the running program with its software architecture model requires an abstraction step. The running program changes its state via changing variable values which is caused by inputs from the environment. This state change is not represented in the SAM. Instead SAM represents the software components and their communication means. Therefore, we require to map changes to communication patterns. For this purpose we introduce the concept of observers. An observer is a piece of software that monitors a certain part of the program’s behavior during the execution. For example, an observer might check whether the number of processes for one software component is equivalent to the specified one. Or an observer checks whether a software component produces a number of events during a certain amount of time. If an observer detects a behavior that contradicts its specification, it computes the appropriate observations in terms of setting the observation predicates $\neg \text{correct}(x)$ for the corresponding connection \textit{x}, and invokes the diagnosis engine.

Definition 3.1 (Observer) An observer is a tuple $(\textit{S}, \Omega)$ with:

1. a set of rules \textit{S} which provides the specification for testing the behavior;
2. a set \textit{\Omega} comprising predicates which correspond to the observations for the SAM model.

The specification of the observer determines it abilities of detecting a misbehavior. An observer for checking the number of processes of a given software component specifies exactly this number. In the current implementation of the observer module we allow to specify the following observers:

- **Periodic event production:** This rule is of the form produces \textit{e} every \textit{n} ms and checks whether \textit{e} is produced at least every \textit{n} milliseconds.
- **Conditional event production:** This rule is of the form produces \textit{e}_1 every \textit{n} ms after \textit{e}_2 and checks whether an event \textit{e}_1 is produced at least every \textit{n} milliseconds after the occurrence of an event \textit{e}_2.
• **Spawn processes**: This rule checks whether a component spawns a number \( n \) of named processes \( id \) and is of the form \( \text{spawns } n \text{ processes } id \).

• **Periodic method calls**: This rule is for checking whether a component calls a method \( m \) at least every \( n \) milliseconds. calls \( m \) every \( n \) ms

The observers are used to monitor the state of the system. For this purpose the observers are implemented and check their rules on a regular basis. In cases of failure the diagnosis procedure is invoked. The following algorithm specifies the monitoring process that has been implemented in our system.

```plaintext
monitoring((CO, C, out, in, WDC, SDC), OS)
Input: The SAM and a set of observer OS.
1. \( M_S = \text{computeModel}((CO, C, out, in, WDC, SDC)) \).
2. \( M_O = \text{computeOSM}odel((CO, C, out, in, WDC, SDC)) \).
3. \( M = M_S \cup M_O \).
4. Do forever:
   a. Let \( OBS \) be the empty set.
   b. For all \( os \in OS \) do:
      i. Check the observer \( os \).
      ii. If \( os = (S, \Omega) \) detects a misbehavior, add \( \Lambda_{os} \cdashrightarrow \Omega \) to \( OBS \).
      iii. Otherwise, add \( \Lambda_{os} \cdashrightarrow \Omega \) to \( OBS \).
   c. If at least one observer detects a failure, call the diagnosis procedure using the model \( M \) and the observations \( OBS \).

Because of the simplicity of the rules monitoring does not take a lot of time. In case of a failure of course the diagnosis procedure has to be invoked which is more time demanding. However, in this case the system is not in a correct state and resources are necessary in order to reset the system. The implementation of the observers sometimes require additional annotations within the original program. In cases where the communication between components is implemented using for example Corba annotations are not required.

### 4 Diagnosis and repair

The diagnosis task in our implementation is based on the model-based diagnosis (MBD) paradigm [Reiter, 1987; de Kleer and Williams, 1987]. In particular we use Reiter’s hitting set algorithm [Reiter, 1987; Greiner et al., 1989] together with a propositional Horn clause theorem prover [Minoux, 1988]. In order to minimize diagnosis time we only search for minimal cardinality diagnoses which can be easily obtained when using Reiter’s algorithm. We only construct the hitting set graph until a level where the first diagnosis is computed. In most practical cases single fault diagnoses can be found. An upper bound for computing single fault diagnosis is determined by the amount of time required for checking consistency. In our case, we have logical rules that can be easily transformed to a set of horn clauses. Hence, time required for checking consistency is of the same order than the number of literals. Because we have to check all single fault diagnoses in the worst case and the size of the model, the worst case diagnosis time is bound by \( O(|C|^2 \cdot |C|^2) \).

After diagnosis those components that are responsible for a detected failure have to be killed and restarted. We have to take care of the fact that restarting one component might require restarting another component. This can be done by using the available information about strong dependencies between components. The components that have a strong dependency relationship with each other have to be restarted. Hence, the steps for repair would be: (1) compute the diagnoses. (2) compute a set of components that have to be restarted. In this step we compute all components that strongly depend on components of a diagnosis. (3) Maximize the chance of repair by using a larger set of components to be restarted. The following algorithm implements this behavior and has to be called by the `monitoring` algorithm in step 4(c).

```plaintext
repair((CO, C, out, in, WDC, SDC), M, OBS)
Input: The SAM, its model \( M \) and observations \( OBS \).
1. Compute diagnoses \( D = \text{diag}(M, CO, OBS) \) where \( D \subseteq CO \).
2. For each diagnosis \( \Delta \in D \) compute the set of strongly dependent components, i.e., \( R(\Delta) = \Delta \cup \{x|3y \in CO : (x, y) \in SDC \vee (y, x) \in SDC\} \). Let \( R = \{R(\Delta)|\Delta \in D\} \).
3. Reduce the set \( R \) by eliminating all elements that are subset of another set in \( R \).
4. Select an element \( x \) from \( R \).
5. Kill all processes that correspond to software components in \( x \). Afterwards restart those processes.

We assume that faults leading to the different diagnoses \( \Delta \) are independent. Whether repair was successful or not in one point in time is detected by the `monitoring` algorithm at a later point in time. Hence, in principle it is possible that repair always tries the same correcting actions without resulting in a correct system state. This problem can be solved either by selecting the components to be restarted (step 4 of repair) non-deterministically or by storing informations about former actions. The latter solution avoids to take the same actions twice.

Figure 2: Timing diagram for diagnosis and repair of a deadlock in the motion service.

### 5 Experimental results

The proposed diagnosis system has been implemented and tested on our mobile robot system. The robot control system
runs on an embedded Pentium III PC with 850 MHz clock rate and 256 MB of RAM. The operating system on the PC is an ordinary Linux system.

The robot control system comprises several software modules. Each module runs as an independent process and implements different services. The services are based on Corba. The communication between these services are implemented either by direct Corba method calls or by an event channel. The diagnosis system itself is implemented as a separate process to minimize the interference with the existing control system. The diagnosis system implements the four types of observers described in Section 3. The use of Corba and OS services allows monitoring the robot control system without any impact to it.

The model of the robot control system (software components, dependencies, observers) are specified in one XML file. Therefore, changes in the model or adaptation to other software systems are simple and straightforward.

The diagnosis system is divided in three modules: a monitoring module (1), a diagnosis kernel (2) and a repair module (3). The monitoring module starts all necessary observers according to the model description and regularly checks for violations of the observers. If such a violation is detected the diagnosis kernel is informed. The diagnosis kernel derives a diagnosis based on the model of the control system and the violated observations. The derivation of a diagnosis is started after a certain amount of time, i.e. 5 s, within no more changes in the state of the observer are detected. This is done for stability reasons as it takes a certain amount of time for all observers to recognize an improper behavior. The diagnosis will be communicated to the repair module. It executes the appropriate repair action to recover the control system. During the repair action no new diagnoses are derived. We do this for stability reasons as the repair action temporally may violate observers. After the repair action is completed the observers and the diagnosis kernel are started again.

For the evaluation of the proposed diagnosis system and its implementation we did several experiments on our mobile robot. We introduced artificial faults into the robot control system and analyzed if the diagnosis system detected and located the fault and recovered the control system. We used two different fault scenarios:

- **Killing a Component**: A certain software component is explicitly killed. This is equivalent to a crash of a certain component.
- **Deadlock a Component**: A deadlock is introduced to a certain software component. This is equivalent to a malfunctioning software component.

Figure 2 shows the timing diagram for the diagnosis and repair of an introduced deadlock in the motion service (MO). After introducing the deadlock in MO the Periodic Event Observer for the event \textit{MotionDelta} perceives that no more events are produced. After the waiting time the diagnosis kernel derived that MO is malfunctioning, \textit{AB(MO)}. Instantly the repair process starts. The repair action comprises a stop of the Behavior Engine (BE), a stop of MO, and a restart of MO and BE. The restart of BE is necessary because BE is strongly coupled with MO. Again after the waiting time the diagnosis kernel derives the diagnosis that all components work properly now. Please note, that no other components were affected by the repair process. The figure also shows the fact that suspending the diagnosis kernel during the repair is necessary as observers report additional improper observations, e.g. Process Observer. The relatively long time for the recovery could be explained by the fact that stopping and starting of services could take a while because of the required starting, stopping and re-configuration of hardware components. The time for computing diagnosis is negligible because it has been less than 10 ms.

Figure 3 shows a more complex scenario. Here we introduce a deadlock in the CAN-Service. After introducing the deadlock, MO and the Sonar Service SO produce no more data because they get no more data from CAN. This fact is perceived by the appropriate observers. Because of the model of the observations, the components and its connections the diagnosis kernel recognizes the malfunctioning CAN. The repair action is similar to the example above except that more
components are involved. After repair, the control system is again in the desired state.

We conducted also two experiments where we killed a software component. In the first experiment we killed the Laser Service (LASER). The diagnosis system successfully detected the fault and recovered the control system by restarting BE, Goal Locator (GL) and LASER. The recovery took 68 s. In a second experiment we killed the World Model (WM). The diagnosis system successfully detected and repaired the fault. During this experiment it was important that the whole process took only 20 s because the system located the fault in the WM and no other component was affected. The affect of the diagnosis system to the runtime performance of the robot control system is negligible. The diagnosis system uses less than 1 % of the CPU time and less than 5 % of the memory.

6 Related research and conclusion

There are many proposed and implemented systems for fault detection and repair in autonomous systems. The Livingston architecture by Williams and colleagues [Williams et al., 1998] was used on the space probe Deep Space One to detect failures in the probe’s hardware and to recover from them. The fault detection and recovery is based on model-based reasoning. Model-based reasoning uses an abstracted logic-based formulation of the system model and the observations. The advantage is that well understood reasoning algorithms could be used. Model-based diagnosis also has been successfully applied for fault detection and localization in digital circuits and car electronics and for software debugging of VHDL programs [Friedrich et al., 1999].

Verma and colleagues [Verma et al., 2004] used particle filter techniques to estimate the state of the robot and its environment. These estimations together with a model of the robot were used to detect faults. The advantage of this approach is that it accounts uncertainties of the robot’s sensing and acting and in its environment because the most probable state is derived from unreliable measurements. Rule-based approaches were proposed by Murphy and Hershberger [Murphy and Hershberger, 1996] to detect failures in sensing and to recover from them. Additional sensor information were used to generate and test hypotheses to explain symptoms resulting from sensing failures. Roumeliotis et. al. [Roumeliotis et al., 1998] used a bank of Kalman filter to track specific failure models and the nominal model. The filter residuals were post-processed to produced a probabilistic interpretation of the operation of the system. Such methods are popular for linear systems affected by Gaussian noise. In [Grosclaude, 2004] a model-based approach for monitoring of component-based software was presented. The behavior of software components were modeled by Petri nets. Places in the net represents the state of a component. Transitions model the interactions with other components. These interactions, sending and receiving of messages, were used to locate a misbehavior in a software component. Liu and Coghill [Liu and Coghill, 2004] used a qualitative representation to model the trajectory of a robot arm. Reasoning about these qualitative trajectories were used to detect and isolate faults of the robot arm.

Previous research has dealt either with hardware diagnosis or diagnosis of software as part of the software engineering cycle. However, diagnosis of software and repair at runtime has never been an issue.

The paper described a model-based diagnosis approach for detecting, locating and repairing software at runtime. For this purpose a modeling technique for representing software architectures which include components, control and data flow, and dependencies between components has been introduced. Moreover, the concept of observers, i.e., software which monitors the activity of the control software, together with their connections to the architecture models have been described in the paper. Finally, the paper presented a repair algorithm and first empirical results of our implementation. These results show that software failures, e.g., deadlocks, can be detected and corrected at runtime.

References


