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

Gerald Steinbauer and **Franz Wotawa**
Institute for Software Technology
Technische Universität Graz
8010 Graz, Inffeldgasse 16b/2, Austria
{steinbauer,wotawa}@ist.tugraz.at

1 Introduction

Control software of autonomous mobile robots is characterized by its fairly high complexity which is in conflict with stability. Complexity is caused by the software components implementing the basic functionality like planning, sensor fusion, actuator interfaces, and their interactions. Because of the high complexity but also the instability of hardware components and connections, and the underlying operating system, complete stability is very unlikely. But if 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.

To achieve this 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. The failure detection is based on observations. For this we use the concept of observers. 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. The fault detection, localization, and correction procedures are all based on declarative models of the control software.

2 Modeling software architectures

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. 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.

The formalization of the structural properties of the software architecture considers the software components, their

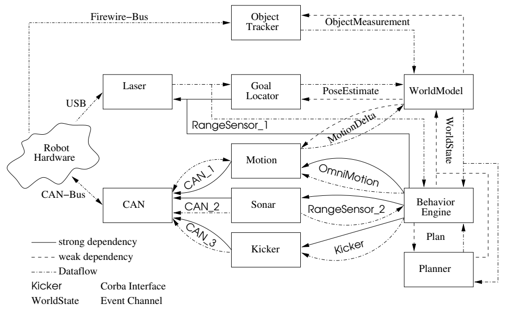

Figure 1: Dependencies between software modules.

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 dependencies between components: weak and strong. Two component are weakly dependent if killing one component directly affects the other component. Otherwise, the relationship is a strong dependency.

A software architecture model (SAM) comprises 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, a set of weak dependencies $WDC \subseteq 2^{CO \times CO}$ and a set of strong dependencies $SDC \subseteq 2^{CO \times CO}$.

The concrete behavior of the software at runtime is determined by the implemented behavior of its software components. An abstraction of the concrete behavior is necessary in order to get a declarative model of the system's behavior. 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.

For example, the rule that represents the abstract behavior of the OT (Object Tracker) component is: $\neg AB(OT) \wedge ok(Firewire) \rightarrow ok(ObjectMeasurement)$ where the predicate AB stands for abnormal, and ok indicates a correct event or method call.

In order to locate root causes, i.e., the components which

[*]This research has been funded in part by the Austrian Science Fund (FWF) under grant P17963-N04.

cause a detected misbehavior, we have to introduce a notation of observations. The easiest way of doing this is to use the same ok predicate for the purpose. If we detect a misbehavior at OM ($ObjectMeasurement$), we could represent this by the literal $\neg ok(OM)$. We introduce a distinguished predicate $correct$ for observations to distinguish observations and computed values . $correct(x)$ for a connection x is true whenever the observed connection shows the correct behavior. Otherwise $correct(x)$ is false. The appropriate model for OT and OM is: $correct(OM) \rightarrow ok(OM)$, $\neg correct(OM) \rightarrow \neg ok(OM)$, $correct(OM) \rightarrow \neg AB(OT)$

3 Monitoring, Diagnosis and Repair

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. But SAM only represents the software components and their communication means. Therefore, we require to map changes to communication patterns. For this purpose we introduce observers. An observer monitors a certain part of the program's behavior during the execution. If an observer detects a behavior that contradicts its specification, it computes the appropriate observations in terms of setting the observation predicates $\neg correct(x)$ for the connection x, and invokes the diagnosis engine. In the current implementation we used the following observers: *Periodic event production* checks whether an event e is produced at least every n ms, *Conditional event production* checks whether an event e_1 is produced n ms after the occurrence of an event e_2, *Spawn processes* checks whether a component spawns a number n of processes and *Periodic method calls* checks whether a component calls a method m at least every n ms.

The observers are used to monitor the state of the system. For this purpose the observers check their rules on a regular basis. In cases of failure the diagnosis procedure is invoked.

The diagnosis task is based on the model-based diagnosis (MBD) paradigm [Reiter, 1987]. In particular we use Reiter's hitting set algorithm [Reiter, 1987]. 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.

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 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 proposed diagnosis system has been tested on our mobile robot system. For the evaluation of the diagnosis 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.

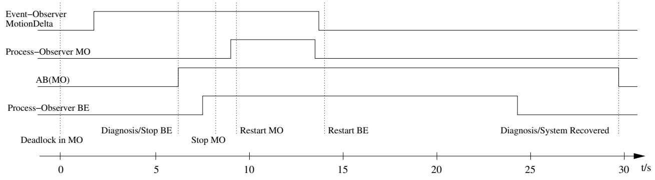

Figure 2: Diagnosis and repair of a fault in the motion service.

Figure 2 shows the results for an introduced deadlock in the motion service (MO). After introducing the deadlock in MO the Event Observer for the event *MotionDelta* perceives that no more events are produced. The diagnosis kernel derived that MO is malfunctioning. 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. After repair the diagnosis kernel derives the diagnosis that all components work properly now. The relatively long time for the recovery could be explained by the fact that a repair of services could take a while because of the required starting, stopping and re-configuration of hardware components.

4 Related research and conclusion

Williams and colleagues [Williams et al., 1998] used model-based reasoning to detect and recover failures of the hardware of a space probe. Verma and colleagues [Verma *et al.*, 2004] used particle filter to estimate the state of the robot and its environment. These estimations together with a model of the robot were used to detect faults. 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 for detecting, locating and repairing faulty software at runtime. For this purpose a modeling technique for representing software architectures which include components, control and data flow, and dependencies has been introduced. Moreover, the concept of observers, 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

[Friedrich *et al.*, 1999] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, 1999.

[Reiter, 1987] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[Verma *et al.*, 2004] V. Verma, G. Gordon, R. Simmons, and S. Thrun. Real-time fault diagnosis. *IEEE Robotics & Automation Magazine*, 11(2):56 – 66, 2004.

[Williams et al., 1998] B. C. Williams et al. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–48, August 1998.