Model-Checking Based Fault Isolation in UML

Dan Lawesson¹, Ulf Nilsson¹, Inger Klein²

¹Department of Computer and Information Science
²Department of Electrical Engineering
Linköpings universitet, SE-581 83 Linköping, Sweden
WWW: http://www.control.isy.liu.se
Email: danla@ida.liu.se, ulfni@ida.liu.se
        inger@isy.liu.se

010106

Report No.: LiTH-ISY-R-2336
Submitted to DX’01

Technical reports from the Automatic Control group in Linköping are available by anonymous ftp at the address ftp.control.isy.liu.se. This report is contained in the file 2336.pdf.
Abstract

We describe an approach to model-based fault isolation in large object oriented control systems. Our method relies on models of structure and behavior expressed in a subset of the Unified Modeling Language (UML). We consider system failures where on-line analysis is problematic, e.g. in safety-critical applications, leading to propagating alarms which have to be analyzed off-line. In our previous work we have shown how to use structural models, in the form of UML class diagrams, to reason about causal dependencies between error messages. In this paper we argue that models of behavior, in the form of UML statecharts, together with verification techniques based on model checking further improve the ability to isolate faults off-line even when both models and observations are incomplete.

Keywords: Fault isolation, object modeling techniques, control system, safety-critical, propagation, model-checking
Model-Checking Based Fault Isolation in UML

Dan Lawesson¹, Ulf Nilsson¹, Inger Klein²
¹Dept of Computer and Information Science
²Dept of Electrical Engineering
Linköping University, 581 83 Linköping, SWEDEN
Email: {danla,ulfni}@ida.liu.se inger@isy.liu.se

Abstract

We describe an approach to model-based fault isolation in large object oriented control systems. Our method relies on models of structure and behavior expressed in a subset of the Unified Modeling Language (UML). We consider system failures where on-line analysis is problematic, e.g. in safety-critical applications, leading to propagating alarms which have to be analyzed off-line. In our previous work we have shown how to use structural models, in the form of UML class diagrams, to reason about causal dependencies between error messages. In this paper we argue that models of behavior, in the form of UML statecharts, together with verification techniques based on model checking further improve the ability to isolate faults off-line even when both models and observations are incomplete.

Introduction

The design of industrial control systems has traditionally been based on functional decomposition but is today complemented, or replaced by object oriented methods. While object orientation has many advantages – most notably support for encapsulation and reuse – it also leads to problems; in the case of system failures an object oriented design is prone to cascading alarms, since an encapsulated object has limited knowledge about the state of other objects, not to speak of the global state of the system.

In this paper we describe a model-based (Hamscher, Console, & de Kleer 1992) approach to fault isolation in large object oriented control systems using a subset of the Unified Modeling Language (UML) (Object Management Group 1999) as modeling notation. The work is motivated by a real industrial robot control system developed by ABB Robotics. The system is large (the order of 10⁶ lines of code), multi-threaded, has an object oriented architecture and is highly configurable, supporting different types of robots and cell configurations. Since the system is time- and safety-critical the first priority, in case of a failure, is to bring the system to a safe state; alarms that go off are logged and can be analyzed when the system comes to a stand-still. We consider here primarily hardware faults, but we assume that the faults show up indirectly in the software. In addition we make the common single fault assumption, i.e. that a system failure is caused by only one fault.

The log thus contains partial information about the events that took place at the approximate time of the system failure. However, the order in which messages are logged does not necessarily reflect the way error messages propagate – the system is multi-threaded and safety critical actions may have to be taken before error reporting takes place. Hence, in what follows we (somewhat conservatively) view the log as a set of error messages. In addition a system may contain a number of critical events that are unobservable, but which may explain all observable alarms.

The ultimate aim of our fault isolation method is to single out the error message that refers to the cause of the failure, or possibly an unobservable critical event explaining the observations. That is, we aim to discard error messages which are definitely effects of other error messages, while trying to isolate error messages (or critical events) which explain all other messages. Given the size of the software it is not possible to use the code directly – we have to rely on a model of the software. The Unified Modeling Language (UML) has become a de-facto standard for modeling object oriented systems using a graphical (and to some extent textual) notation. UML provides a large set of diagrams for describing functional requirements, system structure and behavior (Object Management Group 1999; Wieringa 1998). We use UML class diagrams for modeling the static structure of the software and UML statecharts for modeling behaviors of the software. Relying on UML has the advantage that development of the model is part of the software development process (i.e. it is not required to maintain a separate system model).

In our framework fault isolation proceeds in two phases: In the first phase we use the partial information in the log together with the structural information provided by the UML class-diagrams to build a dependency relation between the error messages; the dependencies are used to discard alarms which are known to be caused by other alarms. We provide a summary of the method in the next section, a detailed account can be found in (Larsson 1999; Larsson et al. 2000).

The structural model is an abstraction of all possible system behaviors; in the case of a complex system the structural information may prove insufficient to isolate the root
cause (due to cyclic dependencies in the static structure).

The second phase of our approach, described in this paper and in greater detail in (Lawesson 2000), relies on a behavioral model (UML statecharts) of the software and a model checker to reason about possible, and impossible system behaviors. That is, we try to infer causal dependencies by reasoning about the temporal order of events. While diagnosis based on a state transition model is not new, see e.g. (Sampath 1995), the use of a model-checker for fault isolation is a novel approach to the best of our knowledge.

The rest of the paper is organized as follows: We begin by summarizing our previous work on fault isolation using structural models. In the following we show how to use behavioral models in the fault isolation process, and finally we summarize and discuss possible extensions of our approach.

**Structural Models**

In this section we outline our previous work on model-based fault isolation with models of static structure. For details see (Larsson et al. 2000).

The input to the method is a log of error messages and UML class diagrams describing the control software. The aim is to construct a dependency graph (called an explanation graph) describing how error messages have propagated through the software. The graph is constructed from information in the error messages and from the class diagrams. There are two types of error messages representing two types of errors. Internal error messages arise when an object detects an internal inconsistency, or a problem with external hardware. Relational error messages are complaints from objects (complainers) requesting services from other object (complainees). Relational error messages can be further divided into those where the complainee is known and where it is unknown. The ultimate goal of the fault isolation scheme is to produce a connected explanation graph without any cycles where all error messages can be traced to a unique maximal error message. This error message is then assumed to be the one closest to the fault. When the dependency graph is not connected the class diagrams can be used to find additional dependencies between objects.

We illustrate our method by an example based on a real scenario from the ABB Robotics control system. The purpose of the example is to illustrate the fault isolation scheme, and we will not go into system details.

In Figure 1, part of the system model of relevance for our fault scenario is shown using UML class diagram notation. The fault considered here is a malfunctioning field bus. To illustrate the use of behavioral models described in the next section we have extended the real fault scenario with an extra object (ibsvme), and a dependency between ibsvme and ibsser in the class diagram (see Figure 1). The resulting error message log is given in Figure 2, and is visualized in a so-called base graph, see Figure 3(a). The original base graph resulting from the error log is shown using solid lines. Each vertex of the base graph corresponds to an object that has either sent an error message (a complainer) or is pointed out by another object (a complainee). The solid edges between vertices correspond to relational error messages and should be read “complains on”. There is also one inheritance relation in the base graph.

The base graph describes dependencies between objects, but the aim is to point out the error message closest to the fault. For this purpose we construct an explanation graph, see Figure 3(b). The explanation graph is in some sense the dual of the base graph; the vertices correspond to error messages and the edges represent dependencies between error messages. The solid lines represent the explanations based on the original base graph.

If the base graph is not connected, as in our case, it may
be necessary to extend the base graph using the UML system model (Figure 1) to produce a connected explanation graph. Algorithms for doing this are further described in (Larsson et al. 1999; Larsson 1999). In our example, an extension on the class level is possible. The basic idea is to try to find complainees for objects in the base graph which have nobody “to blame”.

The system model is searched for associations between classes corresponding to objects in the base graph, possibly via intermediate classes. The result is an extended base graph and corresponding explanation graph as in Figure 3. The extensions are drawn using dashed lines.

In this scenario it is still impossible to obtain a (unique) maximal error message, using only the structural model. A dependency in the structural model, say that \( \text{ibsvme} \) depends on \( \text{ibsser} \) as in Figure 1, means that there may be a scenario where an instance of the class \( \text{ibsvme} \) depends on an instance of the class \( \text{ibsser} \). It is not possible to deduce whether the dependency holds in the scenario at hand, since the model does not discriminate between different scenarios. Hence we cannot deduce whether error message \( 72348 \) or \( 72107 \) is the error message closest to the real fault. To discriminate between them we need additional information.

Behavioral Models

Some of the limitations described above can be alleviated by using a more expressive model, capturing the behavior of the system. Consider again the fault scenario in the previous section. From the structural model we cannot causally order the two error messages \( 72348 \) and \( 72107 \) in Figure 3(b). We show informally how a behavioral model may be used for this purpose.

In Figure 4 we give two simple statecharts for the classes \( \text{ibsvme} \) and \( \text{ibsser} \). Each statechart consists of three states and transitions labeled by events. Events may signal internal state transitions of objects (such as \( 72107 \)) or be used for synchronization of objects, in which case some other object must generate a complementary event (e.g. \( \text{data}! \) and \( \text{data}? \)). An \( \text{ibsser} \) object can send data to an \( \text{ibsvme} \) object which receives the data, does some processing and reacts in one of three ways: (1) it sends a \( \text{ready} \)-signal and returns to the \( \text{idle} \) state, (2) it sends a warning and returns to the \( \text{idle} \) state, and (3) it discovers a serious error (error log message \( 72107 \) – this is a critical event) and enters an \( \text{error} \) state. When \( \text{ibsser} \) receives a warning it issues an error message to the log (error log message \( 72348 \) – this is the second critical event), and then returns to the \( \text{idle} \) state.

If we have error messages from both an \( \text{ibsvme} \) object and an \( \text{ibsser} \) object, it is obvious from Figure 4 that the message from \( \text{ibsvme} \) must have been the last, since message \( 72348 \) cannot follow after \( 72107 \). Now, the single fault assumption and causality (if event \( a \) causes event \( b \), event \( b \) does not precede \( a \)) implies that \( 72348 \) must have caused all other error messages.

Having shown the possible usage of a behavioral model for fault isolation above by example and intuitive arguments, we next describe a mathematical framework and a prototype implementation.

---

2Statecharts also support e.g. hierarchical states, but for brevity we do not consider this extension here.
We first provide the basic semantical foundation of our approach. The semantics of the system is called a **system characterization** and is the set of all system traces, i.e., all possible sequences of events. The semantics of an **observation** and of the properties (predicates) we want to prove is also defined in terms of sets of traces. Due to space limitations, our presentation is simplified. For details, such as inference rules for the semantics of system descriptions and the connection relation, see (Lawesson 2000).

**Events and Traces**

Behavior of the system is described in terms of **events**. An event can either be an **internal** state transition of an object of the system, or a state transition **synchronized** with a state transition of some other object of the system. Since the system consists of a set of objects an event is from now on a pair \( o.e \) consisting of an object \( o \) and a label \( e \) in \( o \)'s statechart.

We reason about a finite set of events \( E = E_{\text{obs}} \cup E_{\text{nobs}} \), where \( E_{\text{obs}} \cap E_{\text{nobs}} = \emptyset \). The events in \( E_{\text{obs}} \) are called **observable** and the events in \( E_{\text{nobs}} \) are called **non observable**. An event is either **normal** or **critical**. The critical events are events that indicate a fault in the system. For a given event \( a \), we write \( \text{crit}(a) \) if \( a \) is a critical event. A finite sequence of events is called a **trace** and a set of traces is referred to as a **system characterization**.

In a given trace, the first critical event that occurs (if any) is called the **root event**. The aim of the fault isolation process is to find the root event.

The set of events of a trace \( T \) is denoted \( \text{events}(T) \). For a given set of events \( A \), we define \( \text{traces}(A) = \{ T \mid \text{events}(T) = A \} \)

In order to reason about root events we use a **system description** and an **observation**. A system description \( SD \) is a set of statecharts – one for each object of the relevant part of the system, together with a **connection relation** that determines how state transitions in different statecharts are synchronized. We do not need a complete behavioral model of the whole system. It is enough with statecharts for the objects that will participate in circular dependencies in the structural model, i.e., the part of the system that cannot be successfully handled by fault isolation using a structural model. In the example above, we need only statecharts for two objects, the ones that have sent error messages that form a cycle in the explanation graph – *ibser* and *ibsome*.

For brevity we assume here that synchronization is possible only between two objects able to perform complementary transitions – transitions labeled \( e! \) and \( e? \). (The connection relation actually provides greater flexibility.) The semantics of the system description \( SD \) defines a system characterization \( m(SD) \), i.e., a set of traces of events. For example, assume that we have a system consisting of an object \( a \) of type *ibsome* and another object \( b \) of type *ibser* and that both objects are initially *idle*. Then a possible execution of the system may look as follows

\[
\begin{align*}
\text{(idle} | | \text{idle}) & \rightarrow \text{b.data!, a.data?} \\
\text{a.ready!, b.ready?} & \rightarrow \text{(process} | | \text{wait)} \\
\text{b.data!, a.data?} & \rightarrow \text{(idle} | | \text{idle)} \\
\text{process} & | | \text{wait)}
\end{align*}
\]

Hence, one possible trace is

\[
\text{b.data!, a.data?}, \text{a.ready!, b.ready?}, \text{b.data!, a.data?}, \text{a.72107}
\]

An observation \( OBS \subseteq E_{\text{obs}} \) is a set of events representing the observed events of a given fault scenario. The model of an observation is defined as

\[
m(OBS) = \bigcup_{O \subseteq E_{\text{nobs}}} \text{traces}(OBS \cup O)
\]

Thus, the semantic interpretation of an observation \( OBS \) contains any number of non observable events interleaved with any permutation and repetition of the observable events in \( OBS \).

To facilitate reasoning about events and traces, we introduce two predicates. For an event \( a \), the property \( \text{present}(a) \) is true in all traces containing \( a \) and the property \( \text{isroot}(a) \) is true in all traces with \( a \) as the first occurrence of a critical event. The semantics of the two predicates is thus defined as follows.

\[
m(\text{present}(a)) = \{ T \mid a \in \text{events}(T) \} \\
m(\text{isroot}(a)) = \{ T \mid \text{crit}(a), \exists i (T(i) = a, \forall j < i \rightarrow \neg \text{crit}(T(j))) \}
\]

Note that for a normal event \( a \), \( \text{crit}(a) \) is false and thus \( m(\text{isroot}(a)) = \emptyset \).

Given a system description \( SD \), an observation \( OBS \) and a predicate \( p \) we define the semantics of entailment and consistency as follows.

\[
\begin{align*}
SD \land OBS & \models p \\
\text{iff} \\
m(SD) \cap m(OBS) & \subseteq m(p) \\
\text{iff} \\
m(SD) \cap m(OBS) \cap m(p) & \neq \emptyset
\end{align*}
\]

If \( \{ SD, OBS, isroot(a) \} \) is consistent we say that \( a \) is an **enabled root** of \( SD \) and \( OBS \). Being an enabled root intuitively means that there is at least one system trace, consistent with the observations, where the given event is the root event. If \( SD \land OBS \models \text{present}(a) \), we say that event \( a \) is **present** of \( SD \) and \( OBS \). Being present intuitively means that the event must appear in all system traces consistent with the observations. If event \( a \) is an enabled root (can be the root) and also present of \( SD \) and \( OBS \) (must have happened), we say that \( a \) is a **strong root candidate** of \( SD \) and \( OBS \). Finally, if \( SD \land OBS \models \text{isroot}(a) \), \( a \) is called the **proven root** of \( SD \) and \( OBS \).

**Reasoning about behaviors**

The predicates introduced above are used to reason about causal dependencies between events in the system in a specific scenario. If the system model was expressed in terms of causality, reasoning about causal relationships in the scenario would be relatively simple, see e.g. (Alur, Peled, &
Logic) is a temporal logic with the following abstract syntax (Clarke, Emerson, & Sistla 1986). CTL (Computation Tree Logic) is a temporal logic with the following abstract syntax (some temporal operators, not needed below, have been omitted)

\[
F ::= A \mid \neg F \lor F \lor \ldots \lor F \rightarrow F \lor AF(F) \lor EX(F) \lor EF(F) \mid EG(F) \lor FA(F)
\]

where \( A \) are atomic (propositional or first order) formulas. A CTL formula is evaluated in a state of a system, and its truth value depends on the possible future executions of that system. For instance:

- \( AG(F) \) holds in a state \( s \) iff \( F \) must hold in all states reachable from \( s \).
- \( AF(F) \) holds in a state \( s \) iff \( F \) must hold in some state reachable from \( s \).
- \( EG(F) \) holds in a state \( s \) iff \( F \) may hold in all states reachable from \( s \).
- \( EF(F) \) holds in a state \( s \) iff \( F \) may hold in some state reachable from \( s \).
- \( EX(F) \) holds in a state \( s \) iff \( F \) may hold in some next state of \( s \).

A CTL formula is said to hold for a system if it holds for the initial state of the system.

The state of our system is the conjunction of its object states. However, such a system state does not in general contain information about the history of events used to reach it. To be able to reason about sequences of events we need also that information. Since the log of events is our observation, we need to be able to discriminate between system states that differ only in the history of events. Therefore, the global system state used when evaluating the following CTL formulas is the conjunction of the object states together with an abstraction of the history of events – a set of events previously observed.

To simplify the presentation, we introduce the following short-hand

\[
obsOK \equiv \forall e \in OBS, seen(e) \land EG(\forall e \in (\text{OBS} - OBS), \neg seen(e))
\]

where \( seen(e) \) is an atomic formula which holds in a global system state if \( e \) is in the set of previously observed events. Intuitively, \( obsOK \) holds in a state if all events in the log have been seen, and it is possible that all other observable events remain unseen.

We say that an event \( e \) is an enabled root if \( \{ SD, OBS, isroot(e) \} \) is consistent. This translates to the following CTL formula

\[
EF(\forall c \in \text{crit} \land \neg seen(c) \land EX(seen(e) \land EF(obsOK)))
\]

Intuitively, an event \( e \) is an enabled root if there is a reachable state \( s_1 \) such that

- no critical event has occurred in \( s_1 \).
- from \( s_1 \) there is a next state \( s_2 \).
- the event \( e \) has occurred in state \( s_2 \).
- a state consistent with the log is reachable from \( s_2 \).

We say that an event \( e \) is a present event if \( SD \land OBS \models \text{present}(e) \). The following CTL formula captures this property.

\[
AG(\text{obsOK} \rightarrow AF(seen(e) \lor \neg obsOK))
\]

Intuitively, an event \( e \) is a present event if whenever the events given in the log are observed, either \( e \) will eventually be seen or some event that is not consistent with the log will take place.

Using the two CTL formulas as input to the SMV model checker, the compiler can perform reasoning about enabled roots, and then about which of the enabled roots that are present events. Events fulfilling both criteria are strong root candidates.

**Examples**

The compiler operates in three phases, each one limiting the set of potential root events. Initially, the set consists of all critical events, and (ideally) after the third phase there is only one event remaining.

During consistency check, it is verified that the given system can give rise to the given observations. During this phase we never remove any members from the set of critical events. The second phase removes all transitions that need some other critical event to be enabled, thus after this phase we only have enabled roots in the set of critical events. In the third phase we remove all non-present events if we have found any that are present. This is equivalent to just keeping strong root candidates if there are any.

We illustrate our approach on four different scenarios:

**I. One Proven Root** If we find an event to be the proven root of the scenario, we are finished and we have the strongest diagnosis result possible with the given model. If we do not find a proven root, we are in one out of three possible situations described in the following sections.
Example 1  In this example we have three objects a, b and c, instances of the same class. Illustrated in Figure 5, the class describes a process that can change between behaving as a client and as a server. When in idle it can either send a call, receive a call or receive a down-message. The message down is sent repeatedly by a process that has been acting as a client and received a fail from its server. The messages down and fail are considered critical, i.e. the possible causes of system failures. The connection relation synchronizes any pair of dual events. One possible trace of the system is

\[
\text{a.call!, b.call?, b.fail!, a.fail!,} \\
\text{a.down!, b.down!, a.down!, c.down?}
\]

Assume that we have observed a.fail! and a.fail?, i.e. we know that object a has both sent and received a fail message.

From this, we can conclude that the first critical event taken by the system must have been a.fail!. Since down is sent after receiving a fail, a down message can never be root when fail messages are critical. Since object a is involved in two transitions of a fail message, the first transmission must have been a.fail! otherwise a would not be available for a second fail transmission. We further observe that a fail transmission leads to the receiving object locking in state err and not participating in anything else than down transmissions. Now, since we only have three objects in the system we know that there can be only two fail transmissions and thus a.fail! was the first.

The input is the system description, observations determining that a.fail! and a.fail? are recorded in the log, and the following list of critical events.

Critical
\[
\text{a.fail!, b.fail!, c.fail!,} \\
\text{a.down!, b.down!, c.down!}
\]

After consistency check all critical events are still candidates and we know that the observations are consistent with the system model. The empty pair of parentheses at the end indicates that no critical events are removed during the first phase.

After consistency check
\[
\text{a.down!, a.fail!, b.fail!,} \\
\text{c.down!, c.fail!, b.down! ()}
\]

In the second phase the compiler finds that there is only one enabled root event, the rest of the events are removed, and displayed within parentheses.

Enabled roots
\[
\text{a.fail! (a.down!, b.fail!,} \\
\text{c.down!, c.fail!, b.down!)}
\]

The third phase becomes trivial – since there is only one root candidate, there is nothing left to do.

Present roots
\[
\text{a.fail! ()}
\]

II. One Strong Root Candidate  If we end up with exactly one strong root candidate, we assume that we have pinpointed the true cause of the fault. This is reasonable to assume, since the event found is the only one that is known to have taken place (its presence is entailed by the scenario) and it is consistent with the given scenario to assume that the event is a root event.

Of course there is still a possibility that there are other enabled root events whose presence are consistent with the scenario, but assuming one of them to be root would demand an explanation to why the strong root candidate (proven to be present!) is not the root.

Example 2  Here we use the same system as described in Example 1, but we let the observations be different. We know that object a has received a fail message, and we know that object c has not sent one. Of course we then know that b.fail! also is in the scenario.

In this case it is consistent to assume that the root event is a.fail!, since if we let object c receive it the observations are still consistent if b sends a fail to a afterwards. It is also consistent to assume that b.fail! is the root event. Therefore, we do not have a proven root in this scenario.

Although we have two enabled roots, only b.fail! is present in the scenario. We therefore keep only the strong root candidate b.fail!.

The first phase accepts the model and observations as consistent, of course no events are removed from the set of candidates.

After consistency check
\[
\text{c.fail!, a.fail!, b.fail!,} \\
\text{c.down!, b.down!, a.down! ()}
\]

In the second phase, the set of candidates is reduced to two events.

Enabled roots
\[
\text{a.fail!, b.fail! (c.fail!,} \\
\text{c.down!, b.down!, a.down!)}
\]

In the third phase the compiler finds that only one of the enabled events is present in the scenario.

Present roots
\[
\text{b.fail! (a.fail!)}
\]

III. No Strong Root Candidate  If the analysis ends with no strong root candidate, we are in a situation where no enabled root is present in the scenario.
Example 3 Here we use the same system as described in Example 1, but we let the observations be different. The observation is very limited – we only know that a has received a fail message.

In this case it is consistent to assume that any of a fail, b.fail! or c.fail! are root events. None of them are present, though. Therefore, we do not have any strong root candidates in this scenario.

The output is the following, where phase two removes the down messages since they are not enabled roots (there has to be a fail before there can be a down). Phase three does not improve the situation, since no critical events are present.

After consistency check
  c.down!, a.fail!, a.down!,
  b.fail!, b.down!, c.fail! ()

Enabled roots
  a.fail!, b.fail!, c.fail! (c.down!, a.down!, b.down!)

Present roots
  a.fail!, b.fail!, c.fail! ()

Perhaps some kind of dependency between the enabled roots can act as guideline for a heuristic approach. For example, if one of the enabled roots entails (i.e. is not only consistent with) the observations that one is a reasonable candidate.

IV. Several Strong Root Candidates If we get two strong root candidates, it becomes very hard to rank them individually. In this case we have two enabled roots that both are known to be present in the scenario.

Example 4 Again, we use the same system as described in Example 1, but we let the observations be different. Now we know that both b and c have sent fail messages.

Now, both b.fail! and c.fail! are enabled roots and present, thus strong root candidates. The scenario is symmetric, though. The two events can have taken place in any order. Therefore we cannot remove any of them from the set of critical events.

The software does not indicate the difference between having more than one strong root candidate and having no strong root candidate. Therefore the output is very similar to the case when there is no strong root candidate.

The only change of the scenario description from the previous examples is the following.

The output is the following, where phase two removes the down messages and a fail!, since it is impossible to have fail! from all objects. Phase three does not improve the situation, since both events are present.

After consistency check
  a.down!, c.fail!, a.fail!,
  b.fail!, b.down!, c.down! ()

Enabled roots
  c.fail!, b.fail! (a.down!,
  a.fail!, b.down!, c.down!)

Present roots
  c.fail!, b.fail! ()

Perhaps some kind of dependency between the enabled roots can act as guideline for a heuristic approach. If the presence of one of the strong root root candidates together with the system description (but without the scenario-specific observation) entails the other strong root events, we have a strong indication that the entailing event is the root.

Integration Fault isolation based on a behavioral model is clearly more expensive computationally and requires more complex modeling than our previous structural approach. To benefit from the advantage of efficient computation of the structural approach we propose a two phase fault isolation schema where structural fault isolation is tried first, and if that is not sufficient for inferring a single root event, we perform behavioral model fault isolation with the remaining candidates as critical events.

Example 5 As discussed previously, the error log in Figure 2 cannot be completely analyzed with the structural approach, but using behavioral information we find a root event. Assume for a more complex example that the system model is larger and that the log contains information that entails presence of one of the critical events. Assume further that the critical events are non observable events.

Now fault isolation becomes more than merely filtering the log of messages, since non observable events might be root events, and must be taken into account. The present critical event is not necessarily the first critical event. The non observable events can be seen as messages that are present in any log, until proven otherwise.

If the log entails $72348$ but not $72107$, it is not possible that the event corresponding to message $72107$ is root, since $72348$ cannot take place after $72107$. Thus, $72348$ is determined to be the root event.

If the log entails message $72107$ but not $72348$, it is consistent with the observations to assume both of the critical events to be root events. Since $72107$ is known to have happened, it is present, we prefer it as the root event. In other words, $72107$ corresponds to a strong root candidate, whereas $72348$ is merely an enabled root.

Conclusions We have shown how models of behavior, in the form of UML statecharts, together with verification techniques based on model checking further improve the ability to isolate faults off-line even when both models and observations are incomplete. Having a behavioral model expressed as state transitions, we are also able to capture and reason about non observable state transitions. For example, there is an opportunity to make our assumption explicit that malfunctioning hardware is manifested in the software. The hardware can be included in the model as simplistic objects with two states, one meaning the component is working correctly, and the other that it has ceased to do so. The transition between the states is the critical event, the real fault that we would like to isolate, but the state transition does not directly give rise to an alarm. It might be possible to deduce that the non
observable event in hardware has taken place using dependen-
tcies in the behavioral model together with an error log from software objects.

It might be argued that the behavioral model used above is only a subset of UML statecharts; in our examples we have used only a limited type of (flat) statecharts, avoiding e.g. hierarchical states, orthogonality and pseudo states. In (Lilius & Porres Paltor 1999) Lilius and Porres provide an operational semantics for UML state diagrams in two steps, where the first steps translates the state diagram to a term rewriting system. Here, the state of an object is a tree (represented as a term) of states with the top state at the root. The hierarchy of the system is captured by the structure of the terms, and orthogonality is also naturally expressed. Pseudo states are needed in the UML notation, but this can easily be expressed in the rewriting system of Lilius and Porres. Thus it is shown that hierarchical states, orthogonality and pseudo states do not add expressivity, and could easily be dealt with also in our approach.

A more serious objection is that our inter-object communication relies on a subset of Harel’s STATEMATE semantics (Harel & Naamad 1996) rather than UML statecharts. From a notational point of view, they look the same, but although under-specified the semantics of UML statecharts differs significantly from Harel’s. In UML events are passed to objects, and put in a dispatch queue, whereas the details about the scheduling of the queue are not specified in the standard. In the STATEMATE semantics of statecharts, an event is merely a handshake or synchronization of two processes. The statecharts used here is a subset of Harel’s, since we require a state transition to be either sending or receiving an event, but never both. This removes the need for micro steps.

A handshake semantics of statecharts has several advantages in our context. First, a handshake models method calls more accurately than any buffered semantics and being the main means of object inter-communication, method invocation is an important factor in fault propagation between objects. Second, since the semantics of UML state-transition diagrams is not formally specified, there is no standardizing benefit from using UML semantics here since such a semantics would have to be invented for this particular purpose. See (Lawesson 2000) for a more detailed discussion about statechart semantics.

An important topic not yet addressed is how to model and reason about structural dynamics, i.e. creation, destruction and mobility of objects; a problem which appears also when trying to model and isolation faults in distributed systems, e.g. ad hoc networks (Fabre et al. 1998). Another topic for future research is to establish design principles to facilitate actually finding a root fault – a problem previously addressed by Sampath (Sampath 1995).

Acknowledgments

This work has been financially supported by NUTEK’s Center of Excellence ISIS – Information Systems for Industrial Control and Supervision. We are also grateful for the cooperation with ABB Robotics, and in particular Magnus Larsson.

References