Towards Behavioral Model Fault Isolation for Object Oriented Control Systems

by

Dan Lawesson

Submitted to the School of Engineering at Linköping University in partial fulfilment of the requirements for degree of Licentiate of Technology

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2000
Towards Behavioral Model Fault Isolation for Object Oriented Control Systems

by

Dan Lawesson

December 2000
ISBN 91-7219-912-1
Linköping Studies in Science and Technology
Thesis No. 863
ISSN 0280-7971
LiU-Tek-Lic-2000:59

ABSTRACT

We use a system model expressed in a subset of the Unified Modeling Language to perform fault isolation in large object oriented control systems. Due to the severity of the failures considered and the safety critical nature of the system we cannot perform fault isolation online. Thus, we perform post mortem fault isolation which has implications in terms of the information available; the temporal order in the error log can not be trusted. In our previous work we have used a structural model for fault isolation. In this thesis we provide a formal framework and a prototype implementation of an approach taking benefit of a behavioral model. This gives opportunities to perform more sophisticated reasoning at the cost of a more detailed system model. We use a model-checker to reason about causal dependencies among the events of the modeled system. The model-checker performs reasoning about temporal dependencies among the events in the system model and the scenario at hand, allowing for conclusions about the causal relation between the events of the scenario. This knowledge can then be transferred to the corresponding fault in the system, allowing us to pinpoint the cause of a system failure among a set of potential causes.

This work has been supported by NUTEK’s Center of Excellence ISIS

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden
Acknowledgments

First and foremost I would like to thank my supervisor Ulf Nilsson for making this thesis possible. Apart from giving me the opportunity to work at TCSLAB, he has given me invaluable guidance and inspiration by being knowledgeable, pedagogical and highly involved in the work. Always available when needed he has been an abundant supply of suggestions of improvements to this thesis at all levels of detail.

I have had the opportunity to work together with Inger Klein and Magnus Larsson of the Automatic Control group of the Department of Electrical Engineering. This cooperation has been very rewarding for me, giving me new insight in the areas of diagnosis and control. Inger has also helped in the writing process by reading and providing plentiful feedback. Thank you Inger and Magnus, without you this thesis would not have been what it is. We have had several interesting discussions with Peter Eriksson and Torgny Brogårdh of ABB Robotics, giving the project real-world motivation.

I appreciate working at TCSLAB, and formerly at LOGRO, and I would like to thank all friends and colleagues that contribute to a friendly and inspiring atmosphere at work.

Lillemor Wallgren, Charlotte Nissen and Gunilla Mellhed have provided much help with administrative duties. Lillemor has also helped planning life as a graduate student, giving structure to the creative and therefore inherently unstructured process of research.

Thank you friends and next of kin for a healthy dose of what is commonly referred to as reality – the part of life that is not work per se, but crucial for maintaining inspiration and energy for work. Finally, thank you Tottis for being encouraging, supporting and my wife.

Linköping, December 2000
Dan Lawesson

This work has been financially supported by NUTEK's Center of Excellence ISIS – Information Systems for Industrial Control and Supervision.
# Contents

1 Introduction .................................................. 1

2 UML and Statecharts ............................................ 5
   2.1 Class diagrams ........................................... 5
   2.2 Statecharts .............................................. 6

3 Discrete Model-based Diagnosis ................................. 9
   3.1 The Discrete Diagnosis Problem .......................... 10
   3.2 Consistency versus Abduction ............................ 10

4 Fault Isolation using a Structural Model ...................... 13
   4.1 Setting .................................................. 13
   4.2 The fault isolation algorithm ............................ 14
   4.3 Limitations .............................................. 17

5 Fault Isolation using a Behavioral Model ...................... 21
   5.1 Events and Traces ........................................ 24
   5.2 Systems and Observations ................................ 24
   5.3 Reasoning ............................................... 25
   5.4 Formal System Description ............................... 25
   5.5 Inference Rules ......................................... 26
   5.6 Causality and Temporal Connections ..................... 27
   5.7 Limitations .............................................. 27

6 A Prototype Implementation ..................................... 29
   6.1 Syntax of the Intermediate Language ..................... 29
       6.1.1 System Classes ..................................... 30
       6.1.2 Object Instances .................................. 30
       6.1.3 Object Connections ................................ 31
       6.1.4 Observations ...................................... 31
       6.1.5 Critical Events .................................... 31
   6.2 SMV representation of system descriptions ............... 32
       6.2.1 Computational Tree Logic .......................... 32
       6.2.2 The SMV modeling language ....................... 33
       6.2.3 Compilation to CTL ............................... 34
CONTENTS

6.3 Computational Efficiency ........................................ 38
6.4 SAD Execution Examples ........................................ 39
  6.4.1 One Proven Root ........................................... 39
  6.4.2 One Strong Root Candidate ................................. 41
  6.4.3 No Strong Root Candidate ................................. 42
  6.4.4 Several Strong Root Candidates ......................... 43

7 Conclusions and Future Work ................................... 45
Chapter 1

Introduction

Object orientation is accepted as a major software design paradigm and the area of application is expanding to new domains. Control systems have traditionally been designed by functional decomposition. The advantage of this approach is that it is closer to the internal design of the computer and therefore lends itself for more simple optimization and analysis of real-time constraints. As the size of the software increases, the well known advantages of object orientation such as modularity and software reuse become more important. Therefore, control systems are now also being developed using object oriented techniques.

One key property of an object oriented system is encapsulation, which is desirable for many reasons but poses a problem when faults occur. In a monolithic piece of software, the entire state of the system can be known at the time of a fault, and an error message issued by the system can be based upon this global system state knowledge. When components of the system are encapsulated in an object oriented system however, it is the intention of the designer to avoid knowledge about the interior state and design of one component to be available to other components. When a fault occurs in this case, an error message can only be based upon the limited knowledge of one component. Thus, if the global state of the system is to be included in the error log the system has to issue several error messages. When one component discovers an error message it will only know part of the whole truth about the system failure and how much error reporting is already taking place. Exceptions are often used in object oriented settings to propagate error messages, but it does not solve this problem since errors propagate along more complicated paths than failed method calls. Therefore, object oriented design may lead to cascading error messages, because of the encapsulation of data and code.

This thesis deals with the problem of performing fault isolation in object oriented control systems. A fault is a defect or a situation that may lead to a failure and a failure is when a system is not exhibiting its intended behavior. We only consider the kind of failure where the system is brought to a halt. Before we can focus on the main task of fault isolation, we have to consider other aspects of the setting.
In a safety-critical control system, fault isolation has to be performed post-mortem unless the fault can be fixed on-the-fly. In case of a critical failure where no automatic recovery is possible, the first priority of the control software is to bring the system to a safe state with e.g. welding equipment switched off and servo engines stopped. When this is done, other tasks such as fault isolation can be performed. This is another reason why simple exception handling would not solve the fault isolation problem – there is a delay between error cause and error message. This, and the fact that the system may consist of several concurrent threads of execution, makes the temporal ordering of messages in the error log less useful and we have therefore chosen to regard the error log as an unordered set of messages.

We have used a control system of ABB Robotics industrial robots as an inspiration. The control system is object oriented, large (in the order of $10^6$ lines of code) and safety-critical, and thus an example of the kind of system that has been outlined so far in this introduction. Furthermore, it is highly configurable with a large set of different supported tools and equipment and several different robots. The same control software is configured to handle a wide range of different settings by installation time configuration. We deal with the critical failures that bring the whole system to a halt. These failures are typically hardware malfunctions or improper user interventions, manifested in the software. We make the common single fault assumption, i.e. we assume that there is no more than one fault causing a system failure. In this thesis, fault isolation refers to the process of taking an error log and a system model as input and providing a minimal set of critical events as output. Critical events are events that can be connected to the cause of system failure and the task of the fault isolation process is to reduce the amount of critical events given in the system model by means of reasoning about the model and the messages in the log. Typically, we prefer to provide a single critical event that transitively explains all the error messages in the log in a causal preorder.

In a setting as outlined, it is reasonable to use a model of the system for performing fault isolation. Since the software in itself is large, it is not feasible to perform analysis on the code itself. An expert system database would become expensive to maintain as software evolves over time and also because of the configurability of the system. Therefore, we advocate a model-based approach. Models are increasingly used in the software development process, and being able to use such a model would decrease the costs of maintaining the model.

In this thesis we present an approach to model-based fault isolation for object oriented control systems. The work is an extension of our previous work (see [9, 10]). Here we use a more sophisticated model of the software, and the result is a more precise fault isolation procedure at the expense of the need for a more detailed software model. The modeling notation used – in this work and in previous work – is a subset of UML.

In our previous work we used a set of rules to perform reasoning mainly about the structure of the software. In this thesis we present an approach that uses a system model containing behavioral information, and to perform reasoning about the behavior of the system we use a model-checker. While
diagnosis based on a state transition model is not new, see e.g. [18] and [20], the use of a model-checker for fault isolation is a novel approach to the best of our knowledge.

The contribution in this thesis is twofold. We provide a mathematical foundation for reasoning about temporal and causal dependencies in order to perform fault isolation in the given setting, and we provide a prototype implementation. The implementation together with the theory constitutes a useful platform for future work on real scenarios and system models.

The rest of this thesis is outlined as follows. In chapter 2 we provide a brief introduction to and pointers to further information about the modeling language UML, the modeling notation used in our work. An outline of the concept of discrete model-based diagnosis can be found in chapter 3. Then, our previous work is outlined in chapter 4. The contribution of this thesis is presented in chapters 5 and 6. The presentation is divided into a formal mathematical description of the problem and its solution in chapter 5 and a prototype implementation in chapter 6. Finally, we outline future work in chapter 7.
Chapter 2

UML and Statecharts

The Object Management Group, OMG see [13], is an organization that provides vendor independent standard specifications supporting analysis and design. OMG was founded in 1989 by eleven companies and is now supported by over 800.

One of the standards proposed and developed by OMG is the Unified Modeling Language, UML see [2, 12, 17]. UML is the amalgamation of the previous main standards for object oriented design and analysis – Booch, OMT andOOSE, with extra expressiveness added.

A primary goal of UML is to enhance further technology and tool development in the software industry by providing a common standard for notation, and many existing software tools (e.g. [14]) and software engineering processes (e.g. [15]) use UML notation.

UML is a large language in the sense that it covers a wide range of topics, such as class diagrams, state-transition diagrams, use cases, collaboration diagrams, sequence diagrams and deployment diagrams. In this thesis we will focus on the class diagrams and the state-transition diagrams.

2.1 Class diagrams

A class diagram illustrates a system design in terms of class details and class inter-dependencies. Apart from having a name, a class contains data and behavior – often referred to as attributes and methods.

Classes relate to each other in several ways. UML contains notation for expressing inheritance, association and aggregation. These are the key concepts used in the structural approach that we present in chapter 4.

Example 2.1 The class diagram in Figure 2.1, showing a part of the control software of ABB industrial robots IO system, illustrates the notation of association and inheritance in a UML class diagram. The class eioexe implements general IO support for other parts of the system, and to achieve this task it uses several devices, implemented in the class eiodev. An instance of the class
**Figure 2.1: An example class diagram**

\( \text{eioexe} \) is associated with several (1 or more) instances of the class \( \text{eiodev} \), which in this case means that an instance of the class \( \text{eioexe} \) uses several instances of the class \( \text{eiodev} \). The class \( \text{eiodev} \) is a generalization of the class \( \text{eiodevIBS} \), or in other terms, \( \text{eiodevIBS} \) inherits (attributes and methods) from \( \text{eiodev} \).

### 2.2 Statecharts

In order to provide a behavioral description of instances of a class, it is possible (but not mandatory) to assign a state diagram to a class. The state diagram describes how an instance of the class – an object – reacts on received events in terms of generating events to other objects as well as performing internal transitions affecting the state of the object.

From a notational point of view, the state diagrams in UML are based on statecharts as presented by Harel in the STATEMATE semantics of statecharts [6], see example 2.2. In this thesis we use the term *statechart* for the statecharts as presented by Harel, and *state diagram* for UML state diagrams.

A statechart is composed of *states* and *transitions*. A state is drawn as a box and transitions are arrows between boxes, the transition points from the source state to the destination state of that particular transition. A state can contain sub-charts that are combined to execute in parallel (called orthogonal composition) or with mutual exclusion (the default composition). There can be actions associated to entering, exiting and also to being in a state. Such actions in states will not be used, and therefore not further discussed in this thesis.

A transition has a trigger event, a boolean guard and an action. When exe-
2.2. STATECHARTS

cution is in the source state, the trigger event is received from the environment and the guard evaluates to true, the transition will be taken and the action will be performed. The action can for example involve sending events to the environment. That is the only kind of action that will be used in this thesis.

Although the semantics of state diagrams in UML is only vaguely standardized, it differs considerably from that of statecharts. 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 UML. In the semantics of statecharts, an event is merely a handshake or synchronization of two processes.

The basic unit of execution of a statechart is called a step. A step takes the system from one state to the next, and involves receiving external stimuli in terms of an event and then reacting to it by internal computing and creation of events. The semantics of statecharts maintains the synchrony hypothesis, the assumption that the system always is fast enough to finish any computing induced by one external event before the arrival of the next external event. This is needed, since one step can consist of a sequence of micro steps. One transition may be enabled by an incoming event, and then create new events which in turn enable other events. According to the synchrony hypothesis, all these causally related transitions, the micro steps, will be computed in the same step.

In this thesis we use a simplified form of statecharts. We do not allow one transition to have both event trigger and action at the same time. That is, a transition is either reception of an event, or execution of an event sending action. This simplifies the step, since it removes the need for micro steps.

Handshake semantics of statecharts has several advantages in our context compared with the UML buffered semantics. 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 of error 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. Third, if needed, buffer functionality can be implemented with a handshaking protocol.

The UML state chart notation is rich, including possibilities of having hierarchical states, orthogonality, memory (by the means of so called history states) and pseudo states such as initial states, join vertices, fork vertices and junctions.

In [11] 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 they 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.

In this work we handle only simple transition systems, but we claim that hierarchical states, orthogonality and pseudo states can be handled by a pre-processor stage in a way similar to the first step described by Lilius and Porres.
in [11]. The history connection available in UML state diagrams is not handled here, nor is dynamic creation of objects and links between them.

**Example 2.2** The statechart in Figure 2.2 is an example (also appearing in [11]) of a statechart with orthogonal and composite states. State $S$ is a composition of sub-states $S_1$, $C$ and $H$. State $S_1$ contains two orthogonal sub-states, $A$ and $B$. When sub-states are orthogonal execution will proceed concurrently in all orthogonal states, but in the case of ordinary composition (as state $A$, containing states $D$ and $F$) the current state is exactly one of the sub-states.

The filled circle with an arrow indicates an initial state, so the initial state for super-state $S$ is $S_1$ (and not $C$ or $H$), and in $S_1$ it is $A$ and $B$ (because of the orthogonal composition). $D$ is initial state of $A$ and $E$ is initial state of $B$. Therefore, initially the state machine is in states $S$, $S_1$, $A$, $B$, $D$ and $E$.

Arrows with associated letters $a$, $b$ and $c$ are labeled transitions. For transition $c$, the source state is $H$ and the destination state is $S_1$. The transition without trigger from $H$ to $S_1$ is taken upon completion of state $H$. Completion of a state is denoted by two concentric circles. When in state $C$ and receiving event $b$, state $S$ is completed.

For example, the initial state in the example can be represented by the term $S(S_1(A(D), B(E)))$ and upon receiving event $a$ the new state becomes $S(C)$. 

![Figure 2.2: An example statechart](image-url)
Chapter 3

Discrete Model-based Diagnosis

When a fault occurs in a composite system, it is important to be able to tell which component or components of the system that have caused the fault. To perform diagnosis is the act of addressing this problem.

The area of diagnosis is thus very wide, ranging over many different applications areas and approaches. In this thesis we focus on model-based diagnosis, and other approaches such as expert systems are not covered.

The research field of model-based diagnosis is divided in two main fields, commonly referred to as FDI and Dx. FDI stands for fault detection and isolation and is the branch of research that has emerged from control theory, whereas Dx stems from computer science. In this thesis we focus on model-based diagnosis with a Dx approach.

When performing model-based discrete diagnosis, the information used is a (discrete) system description and an observation of system behavior. The system description is a model of a system, and the observation is some information about the behavior that the system has exposed in a specific scenario.

The typical domain of Dx model-based diagnosis is (relatively small) static switching circuits modeled on a low level of abstraction, and the observed behavior is the functional correspondence between input and output of the circuit. The models are typically first-order predicate logic formulae, which are very well suited for modeling static switching circuits both in terms of functionality and in terms of fault models – what happens if some component of the system ceases to function properly.

The system consists of a set of components, and a diagnosis is a subset of the set of components such that assuming that the members of the diagnosis are dysfunctional explains the observations.

In the following we give a brief formal definition of the diagnosis problem, and the rest of the chapter contains a survey of model-based discrete diagnosis. A more comprehensive introduction to discrete model-based diagnosis can be
found in e.g. [3].

3.1 The Discrete Diagnosis Problem

Following the terminology from Reiter in [16], we state the diagnosis problem as follows.

A system is a pair \((SD, COMPS)\) where \(SD\) is a system description and \(COMPS\) is a finite set of constants. The intuition is that \(COMPS\) contains names of distinguishable parts of the system and \(SD\) provides a description of the system’s behavior and structure. Each component \(c \in COMPS\) has a set of behavioral modes with semantics defined in \(SD\). To enable reasoning about malfunctioning components, the system description also contains a unary predicate \(AB\), where \(AB(c)\) holds iff component \(c \in COMPS\) is abnormal, i.e. dysfunctional. The behavior of a component \(c\) is naturally connected to \(AB(c)\).

In the most simple form, the system description determines system behavior only in the case of correctly operating components, that is the behavior of component \(c\) is undefined if \(AB(c)\) holds. A more complex system description can include failure modes, that is defining system behavior for component \(c\) for both truth values of \(AB(c)\).

An observation is a set of facts, expressed as first-order sentences. We write \((SD, COMPS, OBS)\) for a system \((SD, COMPS)\) with observations \(OBS\). An observation is often expressed in terms of observed system behavior, for example a relation between input and output signals.

A diagnosis for \((SD, COMPS, OBS)\) is a set \(\delta \subseteq COMPS\) such that the assumption that the components in \(\delta\) are dysfunctional, the diagnosis hypothesis, explains \(OBS\).

There are two main interpretations of what to require from a diagnosis in order to say that the diagnosis hypothesis explains a given observation – the strong abduction-based diagnosis and the weaker consistency-based diagnosis.

3.2 Consistency versus Abduction

There are two main approaches to model-based diagnosis: abduction-based diagnosis and consistency-based diagnosis. The weaker consistency-based approach, designed to handle the case when only a model of correct system behavior is available, only demands the diagnosis hypothesis to be consistent with the observations and the system description. Abduction-based diagnosis was originally proposed as a way to handle a richer system model covering correct and faulty system behavior. Here the diagnosis (together with the system description) is required to entail the observations.

More formally, a consistency-based diagnosis for \((SD, COMPS, OBS)\) is a minimal set \(\delta \subseteq COMPS\) such that

\[
SD \cup OBS \cup \{AB(c) | c \in \delta\} \cup \{\neg AB(c) | c \in (COMPS - \delta)\}
\]
**3.2. CONSISTENCY VERSUS ABDUCTION**

is consistent.

An *abduction-based diagnosis* for \((SD, COMPS, OBS)\) is a minimal set \(\delta\) of behavioral mode assumptions such that

\[
SD \cup \delta \models OBS
\]

and

\[
SD \cup \delta \text{ is consistent.}
\]

In the abduction-based approach, since the system model (together with the diagnosis) should entail the observations, it has to contain an exhaustive model of all faults that should be handled. This naturally yields a very strong diagnosis, but at the expense of demanding an extensive system model. This approach does not work well in an incomplete or a non-deterministic setting. If the modeled system can behave in a non-deterministic manner, a particular set of observations will not be entailed.

The consistency-based approach does not suffer from the same limitation, since here a diagnosis only needs to be consistent with the model and the observations. Thus, we are able to compute a diagnosis without an explicit exhaustive fault model. Having relaxed the notion of diagnosis, we naturally obtain a weaker diagnosis, or often several possible diagnoses.

As well as the abduction-based diagnosis benefits from fault models, one can perform consistency-based diagnosis with the additional information of a fault model. In this case, the two approaches become very similar and unified frameworks have been proposed, e.g. [4, 21].
Chapter 4

Fault Isolation using a Structural Model

In our previous work, see [9, 10], we have investigated model-based fault isolation of object-oriented control systems using a structural system model. In this chapter we summarize the results and conclusions from this work which forms the basis of the rest of this thesis. Further details can be found in the technical report [9] and in the PhD thesis by Larsson [8].

4.1 Setting

A large object oriented control system from ABB Robotics is used as a case study. The system studied is safety critical, and thus the first priority in case of a failure that cannot be handled on-the-fly is to bring the system to a safe state. No other activities can begin until this is achieved. Therefore, we have to perform post-mortem fault isolation.

Since the system is highly configurable, and the software evolves, it would be costly to maintain an expert system database. A model-based approach is better suited for abstracting away from configuration details and as the model is updated along with the software itself the cost for maintaining the model is low.

Thus, the setting implies a post mortem model-based fault isolation approach. Post mortem model-based fault isolation is a concept closely related to model-based diagnosis, as described in chapter 3. Here we also have a model of the system software, the error log resembles partial observations in terms of faults in components, and we aim to find a component to blame the failure on. The domain is not that of classical discrete diagnosis, though. In this work we model the software and thus the components in the diagnosis are software entities. The model has a high level of abstraction and encompasses a large dynamic software system, whereas the classical domain for diagnosis is static switching circuits described in detail at the level of abstraction of electric gates.
and switches. In the classical diagnosis case we have complete information about system behavior, whereas in our case we are limited to the partial information in the error log.

The result of the fault isolation is semantically closely related to that of a diagnosis – we get a (hopefully singleton) set of error messages, pointing to components that we assume to be faulty. The system model is a model of the correct behavior of the system, therefore the situation is analogous to a consistency-based diagnosis.

4.2 The fault isolation algorithm

We have developed a tool, called DrRobot, that takes an error log and a system model as input and produces a filtered log as output. Ideally, the filtered log contains exactly one message. This outline of the algorithm is only intended to give an intuition of how DrRobot works and is therefore simplified. More details can be found in [9, 8], while the presentation in [10] gives an overview.

The input accepted by DrRobot is a software system model in the form of UML class diagrams and an error log in the form of a text file. Associated with each error message in the log there is a characterization containing information about complainer and complainee. The complainer is the object that issued the error message, and the complainee is the object that the complainer has pointed out as culprit, if any.

If the origin of the error message is contained within the responsibility of an object, e.g. the object is the software driver for malfunctioning hardware, we call the error message internal. If, on the other hand, the error message indicates the malfunction of some other object in the software system, the error message is relational. If it is relational, the complainee is either known or not known. In case the error is related to inter-thread communication, there are more possible characterizations, see [9, 8] for details.

If the dependencies of error messages in the log do not suffice to determine one error message that explains all the other, the system model is used. The basic idea is to use dependencies between classes manifested in the system model to draw conclusions about run-time dependencies between the objects that have reported the error messages in the log. For example, if in the UML model class A is associated with class B via a "uses"-relation, DrRobot assumes that object a depends on object b if a is an instance of A and b is an instance of B.

The dependency relation between objects is then interpreted as an explanation relation between corresponding objects such that if object a depends on object b, an error message from object a is explained by an error message from object b. Other relations between classes, such as packets and inter-thread communication are also used, but not further described here.

The explanation relation on the error messages can be used as output in itself, but since the aim is to produce a filtered log, the output is the set of maximal elements of the "explained by" relation on the error messages. If the model is strong enough and the scenario is benign, the relation will have a
THE FAULT ISOLATION ALGORITHM

8. 71061 I/O bus error 0105 13:45.30
   Reason:
   - An abnormal rate of errors on bus IBS has been detected.
   Relational: eibus->eioexe

9. 71107 InterBus-S bus failure 0105 13:45.31
   Reason:
   - Lost contact at address 2.3
   Internal

10. 71139 Access error from IO 0105 13:45.35
    Reason:
    - Cannot Read or Write signal DO3_1 due to communication down.
    Relational: eio->eiount

11. 40503 Reference error 0105 13:45.35
    Device descriptor is not valid for a digital write operation
    Relational: rlio->eio

12. 40223 Execution error 0105 13:45.35
    Task MAIN: Fatal runtime error
    Relational: pgmexe->RealInstruction

Figure 4.1: An example error log, annotated with error message characterizations. If relational, we write complainer->complainee.

The main classes relevant for this example are the following. The class pgmexe has responsibility for execution of the robot control language RAPID, and instructions of a RAPID program have the common ancestor RealInstruction. One of the instructions that handle IO is called rlio, which in turn uses the implementation of Elementary IO in the class eio. The class eiount contains specific code for EIO-units, used by eio, and eiount uses driver specific functionality provided by the class eioexe. The class ibsser contains

greatest element, i.e. one error message that transitivity explains all the others.

Example 4.1 In this example, also appearing in [10], the input to DrRobot is the error log that appears in a simplified form in Figure 4.1 and a UML system model containing, but not limited to, the class diagram in Figure 4.2 showing relations between classes dealing with bus communication and execution of customer robot programs. The error log comes from a real fault scenario, with some irrelevant error messages removed and error message characterizations added. In the real world implementation there is a database containing a mapping from error message numbers to message characterizations, thus the information depicted in the log here is the union of the relevant parts of this database and the error log.
Using the error message characterization in the error log and inheritance information from the system model, DrRobot constructs a dependency graph between classes called the base graph depicted in Figure 4.3. Here, objects are vertices and error messages are directed edges. The dashed boxes are packets, i.e. groups of tightly cooperating objects. Error message 71107 is an internal error message, i.e. the complaining object knows that there is no complainee. To indicate this, the edge is self-referring.

Now, it is clear that error message 40223 can be explained by error message 40503, since the complainee of the former is a superclass of the complainer of the latter. Similarly 71139 is weakly explained by 71061 since the complainee of the former is in the same package as the complainer of the latter. The intuition behind this explanation is that since the complainee of 71139 and complainer of 71061 are in the same package, they are semantically coupled and error propagation between them is likely although not explicit in the model. Since the dependency is inferred from a rather weak ad hoc rule, the explanation is considered weaker than explanations originating from explicit modeling constructs. Rules such as these, and additional ones, are discussed in detail in our previous work [9, 10]. The “explained by”-relation can be depicted in the explanation graph of Figure 4.4 - the dual of the base graph in the sense that messages are vertices instead of edges, and it is evident that error message 71107 transitively explains all other error messages. Dashed edges indicate weak explanations.

Using class diagram information, DrRobot computes the extended base graph shown in Figure 4.5. Figure 4.2 depicts a small part of the class diagram describing the software. As seen, ibser is used by eiodevIBS, which is a eiodev, that in turn is used by eioexe. Similarly, there is a dependency from eiount
Figure 4.3: The base graph

to eioexe to be found elsewhere – not included in Figure 4.2 – in the class diagram. Since eiodevIBS was not present in the original base graph, it is drawn in a dashed box in the extended base graph. By including eiodevIBS in the extended base graph, DrRobot can strengthen the explanation graph. The new graph is shown in Figure 4.6 where all error messages are strongly explained by 71107.

4.3 Limitations

Since the system model used is static and structural, the dependencies between classes in the system are the same no matter what we know about the present scenario. A relationship between two classes in the class diagram determines that the classes exhibit a dependency in some scenario, and is therefore used in all scenarios.
This problem does not only make the fault isolation imprecise, it can also make it impossible in less benign cases. Sometimes the structural model, which is a model of all possible executions, contains cyclic dependencies that are never present in any scenario. In that case the structural fault isolation cannot infer anything about the dependencies between the objects in the cycle.

Another limitation is inherent in the model abstraction from objects to classes. The structural approach using class diagrams relies on the assumption that there is a one-to-one mapping from objects participating in the error log to the classes of the software. If several instances of the same class participate in a scenario, no part of the structural class system model can be used in fault isolation. This is not a problem in the ABB case, since the system is designed in a way where the main actors of a scenario are unique instances of their respective classes, but it has not been investigated whether this is a general rule for control systems.
Figure 4.5: The extended base graph
Figure 4.6: The second explanation graph
Chapter 5

Fault Isolation using a Behavioral Model

In chapter 4 we presented previous work on model-based fault isolation for object-oriented control systems using a structural system model. The main topic of this thesis is fault isolation using a behavioral system model, and this chapter together with chapter 6 are dedicated to that subject.

We claim that some of the limitations described in section 4.3 can be alleviated by using a more expressive model. The approach proposed in this thesis is to use a model that not only captures the structure of the system, but also contains information about the behavior of the objects of the system.

A dependency in the structural model, say class A depends on class B, means that there exists a scenario\(^1\) where an instance of class A depends on an instance of class B. It is not possible to deduce whether the dependency holds in the scenario at hand or not, since the model does not discriminate between neither different scenarios nor different instances of a class. By modeling also the behavior of the objects we get a chance to reason about dependencies that hold only under certain circumstances, i.e. in certain scenarios.

**Example 5.1** As a motivating example, consider again the fault scenario in Example 4.1. Assume that there is no internal error message, but instead a cycle of two messages, where 72348 goes from ibsser to ibsvme (a new object, introduced for this example) and error message 72107 is directed in the opposite direction as shown in Figure 5.1.

Using the structural model we cannot discriminate between the two error messages 72348 and 72107. We show how a behavioral model in some cases can be used for this purpose. Assume that the classes ibsvme and ibsser were equipped with the statecharts in Figure 5.2.

An ibsser object can send data to an ibsvme object which receives the data and performs some processing after which it reacts in one of three ways: (1) it

\(^1\)Here we use the term "scenario" to denote a particular execution of the system.
CHAPTER 5. FAULT ISOLATION USING A BEHAVIORAL MODEL

Figure 5.1: The base graph

sends a ready-signal and returns to the idle state, or (2) it sends a warning and returns to the idle state, or (3) it discovers a serious error (error log message 72107) and enters an error state. When ibsser receives a warning it issues an error message to the log (error log message 72348), and then returns to the idle state.

If we have error messages from both an ibsvme object and an ibsser object, it is obvious from Figure 5.2 that the message from ibsvme must have been the last, since message 72107 cannot follow after 72348. Now, the single fault assumption and causality (if event $a$ causes event $b$, event $b$ does not precede $a$) implies that 72107 caused all other error messages.

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 that 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 error message. It might be possible to deduce that the non-observable event in hardware has taken place using dependencies in the behavioral model together with an error log from software objects.

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

We begin by introducing traces and system characterizations. This will be the basic semantical foundation for the rest of the chapter. The semantics of a system model is defined in terms of a system characterization containing all the system traces, i.e. all possible behaviors. The semantics of an "observation" and of the predicates we use is also defined in terms of sets of traces.

Consistency and entailment will be central logic tools for the reasoning. Consistency is used to express possibilities of some events to have happened whereas entailment expresses the necessity of a series of events to have taken place. Using traces, reasoning in terms of consistency and entailment relates to set operations. For instance, if the intersection of the semantics of a system model and a given observation is non empty, the system model and the observations are consistent, i.e. it is possible that the system modeled exhibits the given observations. In a similar way, entailment corresponds to set inclusion.

Since we want a syntax of system models that is a subset of statecharts notation (and of UML state-transitions diagrams), we need to define the operational semantics of the used subset. This is done by providing inference rules, which define a labeled transition system which has a clear semantics in terms of sets of traces, i.e. system characterizations.

![Figure 5.2: State machines for ibsvme (left) and ibsser (right).](image-url)
5.1 Events and Traces

Behavior of the system is described in terms of events. An event can be an internal state transition of an object of the system, or a state transition that is synchronized with a state transition of some other object of the system.

We reason about events from a finite set $\mathcal{E} = \mathcal{E}_{\text{obs}} \cup \mathcal{E}_{\text{nobs}}$, where $\mathcal{E}_{\text{obs}} \cap \mathcal{E}_{\text{nobs}} = \emptyset$. The events in $\mathcal{E}_{\text{obs}}$ are observable and the events in $\mathcal{E}_{\text{nobs}}$ are non-observable. An event is either a normal event or a critical event. For a given event $a$, this is reflected by the predicate $\text{crit}(a)$ that holds iff $a$ 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.

The critical events are events that indicate a fault in the system. In a given scenario, the first critical event that occurs is called the root event. The aim of the fault isolation process is to find the root event.

A trace containing $n$ events is written $[a_1, a_2, ..., a_n]$. Given a trace $T$ we use $T(i)$ to refer to the $i$th element in $T$, i.e., if $T = [a_1, a_2, a_3, ..., a_n]$ then $T(i) = a_i$.

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

Example 5.2 Consider a set of events $A = \{ a_1, a_2 \}$. Then $\text{traces}(A)$ is an infinite set of traces containing exactly the two events $a_1$ and $a_2$ in any order and any number of duplicates. That is $[a_1, a_2] \in \text{traces}(A)$ and $[a_2, a_2, a_1, a_2] \in \text{traces}(A)$, but $[a_1] \notin \text{traces}(A)$.

5.2 Systems and Observations

In order to reason about root events we use a system description and a set of system observations.

A system description $SD$ is a syntactic description of system behaviors. The semantics of the modeling language defines a system characterization $ST = \text{mod}(SD)$ which is a model of $SD$ and therefore also of the system. In Section 5.4 we define $\text{mod}(SD)$ in terms of the set of traces that the system can exhibit.

An observation $OBS \subseteq \mathcal{E}_{\text{obs}}$ is a set of events that represents the observed events of a given fault scenario. As pointed out above the reason for treating the observations as a set instead of an ordered sequence is that the intended application is safety critical, and thus there might be limited correlation between ordering of detection of errors and error messages. The error reporting component may have to perform other safety related tasks before it can report anything to the error log. Furthermore, the system is a set of concurrently operating objects and the messages from different threads of execution are interleaved in the log.

The model $\text{mod}(OBS)$ of an observation $OBS$ is defined as

$$\text{mod}(OBS) = \bigcup_{O \subseteq \mathcal{E}_{\text{nobs}}} \text{traces}(OBS \cup O)$$
Example 5.3 Consider the observation \( \text{OBS} = \{ a_1, a_2 \} \) and assume that \( u \) is the only non-observable event, i.e. \( \mathcal{E}_{\text{obs}} = \{ u \} \). Then \( \text{mod(OBS)} \) is an infinite set of traces containing \( a_1 \) and \( a_2 \) in any order, and possibly interleaved with \( u \). I.e. \( \text{traces}(\{a_1, a_2\}) \subset \text{mod(OBS)} \) and \( [u, a_1, u, a_2, u, u] \in \text{mod(OBS)} \).

5.3 Reasoning

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 critical event. The semantics of the two predicates is thus defined as follows.

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

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

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

\[
SD \wedge OBS \models p \iff \text{mod(SD)} \cap \text{mod(OBS)} \subset \text{mod(p)}
\]

\[
\{ SD, OBS, p \} \text{ is consistent } \iff \text{mod(SD)} \cap \text{mod(OBS)} \cap \text{mod(p)} \neq \emptyset
\]

If \( \{ SD, OBS, \text{isroot}(a) \} \) is consistent we say that \( a \) is an \textit{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 \( SD \wedge OBS \models \text{present}(a) \), we say that event \( a \) is \textit{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 \textit{strong root candidate} of \( SD \) and \( OBS \). Finally, if \( SD \wedge OBS \models \text{isroot}(a) \), \( a \) is called the \textit{proven root} of \( SD \) and \( OBS \).

5.4 Formal System Description

In this section we give a formal definition of a system description \( SD \), and provide semantics in terms of a system characterization \( ST = \text{mod(SD)} \).

A system description \( SD \) is a tuple \( \langle O, C, cl, \rangle \), where

- \( O \) is a finite set of objects denoted \( o \)
- \( C \) is a finite set of classes \( c_i = \langle S_i, L_i, \rightarrow_i \rangle \) where
  - \( S_i \) is a finite set of states denoted \( s \)
  - \( L_i \) is a finite set of event labels denoted \( \alpha \) or \( \beta \).
  - \( \rightarrow_i \subseteq S_i \times L_i \times S_i \) is a finite set of labeled transitions
CHAPTER 5. FAULT ISOLATION USING A BEHAVIORAL MODEL

Given $c_i = \langle S_i, L_i, \rightarrow_i \rangle$ we use the notation $S(c_i) = S_i$, $L(c_i) = L_i$ and $\rightarrow_{c_i} = \rightarrow_i$. Instead of writing $\langle s_s, \alpha, s_d \rangle \in \rightarrow$ we write $s_s \xrightarrow{\alpha} s_d$.

• $cl : O \rightarrow C$. We use the notational convention $S(o) = S(cl(o))$, $L(o) = L(cl(o))$ and $\rightarrow_o = \rightarrow_{cl(o)}$.

• $\triangleright \subseteq E \times E$ where $E = O \times L(O)$, with the restriction that $o.\alpha \in E$ only if $\alpha \in L(o)$. A pair $a \in E$ is called an event and the intuitive interpretation of $a \triangleright b$ is that $a$ and $b$ are synchronized events, with $a$ as sender and $b$ as receiver. Therefore, we use the term connection relation to refer to $\triangleright$.

A system is a set of communicating objects, and a (global) system state is a mapping $q : O \rightarrow S(O)$ i.e. $q = \{ o_i \mapsto s_i \}_{o_i \in O}$ where $s_i \in S(o_i)$. $Q(SD)$ denotes the set of all global states for SD. When SD is clear from the context we write simply $Q$.

5.5 Inference Rules

We define some inference rules for system state transitions, in order to provide a labeled transition system semantics for the system. The inference relation of course depends on $SD$, but to simplify the syntax we let the corresponding $SD$ be determined by the context where the inference rule is used. The operator $\sqcup$ is used for disjoint union, i.e. when writing $A \sqcup B$ we assume that $A$ and $B$ are disjoint.

The relation $\triangleright$ defines connections for handshakes between objects.

\[
S \sqcup \{ o_1 \mapsto s_1, o_2 \mapsto s_2 \} \xrightarrow{\alpha \triangleright \beta} S \sqcup \{ o_1 \mapsto s'_1, o_2 \mapsto s'_2 \}
\]

if $\alpha \triangleright \beta$

Internal events, i.e. events that can take place without any handshake with another object, are indicated by loops in the connection relation.

\[
S \sqcup \{ o \mapsto s_s \} \xrightarrow{\alpha \triangleright \alpha} S \sqcup \{ o \mapsto s_d \}
\]

if $\alpha \triangleright \alpha$

Given a system description $SD = \langle O, C, cl, \triangleright \rangle$ the inference rules above induce a labeled transition system $\langle Q, \rightarrow \rangle$, that provides the behavioral semantics for $SD$.

Now we are ready to define $ST = mod(SD)$, the behavioral meaning of the system description $SD$. Let $\langle Q, \rightarrow \rangle$ be the labeled transition system induced by $SD$, then $ST$ is a set of event label sequences, defined as follows.

\[
ST = \{ \langle a_1, b_1, ..., a_n, b_n \rangle \mid \exists s_0, ..., s_n (s_0 \xrightarrow{a_1} s_1 \Rightarrow ... a_n \xrightarrow{b_n} s_n) \}
\]
5.6 Causality and Temporal Connections

The predicates introduced in section 5.3 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. [1] for an example of such a model. In this work, though, we have chosen to use a temporal model of the system since that is the type of model that can be more easily incorporated in the software development process. There is therefore a need for extracting causal dependencies from a temporal model.

In principal, we have one major connection between the concepts of causality and temporal order, namely the basic fact that for one event to cause another event, the causing event must not occur after that caused. The temporal ordering of events thus puts a necessary condition on the causal ordering.

5.7 Limitations

As discussed in Section 2.2, we use a limited but still expressive subset of state-charts with semantics in terms of synchronization rather than dispatch queues. This approach, where we do not adhere to the UML standard is motivated, as explained in Section 2.2, by the need for modeling of synchronization since that is an important factor in error propagation.

We do not handle dynamic creation of neither objects nor links between objects. Thus, mobility and systems with dynamic topology cannot be modeled. In this work we use a model-checking tool that operates on a finite static transition relation, therefore boundless mobile systems and system with dynamic topology cannot be modeled.
Chapter 6

A Prototype Implementation

We have developed a prototype system, SAD\textsuperscript{1}, that finds strong root candidates given a description of a system and an observation. SAD uses the model-checking tool SMV, see [7], for reasoning about faults and alarms in object-oriented control systems. Section 6.2 contains a description of the approach. Statecharts is a graphical notation and the model specification language used for input to SAD is intended to be an intermediate textual language. The syntax used is described in Section 6.1, and Section 6.4 contains a description by means of examples of the way in which SAD works.

6.1 Syntax of the Intermediate Language

In this section we describe the syntax of the intermediate language for system descriptions and observations, and relate it to the semantics described in section 5.4. We would like to be able to analyze models expressed in UML state diagrams notation, and therefore we have chosen a syntax intended to express transitions in a way similar to state diagrams, making translation from UML state diagrams to our representation as simple as possible. The main difference in the semantics between our representation and UML state transition diagrams is that here we regard events as handshakes (i.e. synchronization) instead of having communication channels with buffering capacity as in UML.

An input file consists of sections. There are five different kinds of sections, recognized by keywords. The following table maps keywords to a short description and a reference to the section describing the details.

\begin{itemize}
\item \textbf{SAD} is a second phase addition to DrRobot, described in Chapter 4.
\end{itemize}
6.1.1 System Classes

In a class section, a set of transitions is associated with a class name. A transition is on the form $\text{start}-\text{evt}-%\text{end}$ where $\text{start}$ is the source state and $\text{end}$ is the destination state of the transition, while $\text{evt}$ is an event label. The source state of the first transition in the list is the initial state of the class.

There are three kinds of event labels, message sending, message receiving and internal events. A question mark indicates message reception and an exclamation mark indicates messages sending. In one transition, at most one message can be either sent or received. If the event label does not end in neither a question mark nor an exclamation mark, it denotes an internal event. An internal event can be labeled with the empty string, in which case the compiler chooses a unique name from the sequence $1, 2, 3, \ldots$, but in this case the event cannot be referred to when describing observations and critical events (see sections 6.1.4 and 6.1.5).

Example 6.1 The following is an example of a server that accepts requests, processes them and then sends ack if processing completes without failure. The initial state is start.

```plaintext
class Server
  start --> idle
  idle - req? -> process
  process - ack! --> idle
  process - fail --> failed
```

The corresponding class description $<S, L, \rightarrow> \in C$ looks as follows.

$$
S = \{ \text{start, idle, process, failed} \} \\
L = \{ \tau_1, \text{fail, req?}, \text{ack!} \} \\
\rightarrow = \{ \text{start} \xrightarrow{\tau_1} \text{idle, idle} \xrightarrow{\text{req?}} \text{process, process} \xrightarrow{\text{ack!}} \text{idle, process} \xrightarrow{\text{fail}} \text{failed} \}
$$

Since the transition $\text{start} \xrightarrow{\tau_1} \text{idle}$ does not require any handshaking we conclude that $o.\tau_1 > o.\tau_1$ for any object $o$ of class Server.

6.1.2 Object Instances

The classes described in section 6.1.1 are mere templates that can be used to instantiate arbitrarily many objects. Instances are declared by mapping an object name to a class name.
Example 6.2 Below we define two objects.

instances
  s : Server
  c : Client

6.1.3 Object Connections

The structure of the system, in terms of object inter-connections is described by defining connections for message passing. Basically the section of connections is a list of the same format as in \( \triangleright \), but for convenience we allow for some abbreviations. We drop ? and !, since the direction of the arrow contains information about which event is sending. By omitting labels we get all connections that have compatible labels, e.g. \( s \rightarrow c \) yields all connects on the form \( s.<\text{label}!\rightarrow c.<\text{label}>? \) allowed by the corresponding class definitions. Arrows in both directions expand to two statements, one in each direction, e.g. \( s\leftarrow c \) yields \( s\rightarrow c \) and \( c\rightarrow s \) which then will be further macro expanded by the first rule.

Example 6.3 Here we define \( \triangleright = \{ (c.req!,s.req?), (s.ack!,c.ack?) \} \).

connects
  c.req\rightarrow s.req
  s.ack\rightarrow c.ack

6.1.4 Observations

An event \( a \in \mathcal{E} \) is either observable, \( a \in \mathcal{E}_{\text{obs}} \) or non-observable, \( a \in \mathcal{E}_{\text{nobs}} \). For a specific observation, a set \( OBS \subseteq \mathcal{E}_{\text{obs}} \) has been recorded in a log. Apart from defining \( OBS \) the section of the language regarding observations also defines \( \mathcal{E}_{\text{obs}} \). The section contains a comma-separated list of events, each event possibly prefixed by an exclamation mark. Events not listed are assumed to be non-observable, events prefixed by an exclamation mark are not recorded in the log, whereas the others are.

Example 6.4 In this example, \( \mathcal{E}_{\text{obs}} = \{ s.ack!, c.fail \} \) and \( OBS = \{ s.ack! \} \).

observations
  s.ack!, !c.fail

6.1.5 Critical Events

This part is a comma separated list of the events that are critical events, that is potential root events.

Example 6.5 Here we define two critical events.

critical
  s.fail, c.fail
6.2 SMV representation of system descriptions

Given a system description and an observation in the syntax presented previously in this chapter and with semantics presented in chapter 5, SAD uses an existing model-checking tool to reason about the predicates introduced in section 5.3 in order to find a minimal set of root events.

The model-checking tool used is SMV [7] which uses a symbolic model-checking algorithm to evaluate Computational Tree Logic [3] (CTL) formulas. The model is presented to SMV in the form of a specification language that is similar to the language used previously in this chapter. The task of SAD is to translate the system description to a model in the language accepted by SMV and to express the predicates in terms of CTL formulas.

6.2.1 Computational Tree Logic

Assume that we have a system behavior described by a transition relation that can be depicted as a graph with states as vertices and state transitions as edges. The computation tree is the unwinding of the state transition graph to a directed tree with the initial state as root. The children of a state in the computation tree are all immediate successor states. The paths from the root of a computation tree represent all possible computations of the transition relation. We only consider infinite computations here, by assuming that the leaves of the computation tree have self-referring loops. The states are described by state predicates that capture relevant properties of the states. For a state predicate $P$ and a state $q$ we write $q \models P$ when $P$ holds for state $q$.

The syntax of CTL formulae can be defined inductively as follows.

- All state predicates are CTL formulas.
- If $f_1$ and $f_2$ are CTL formulas, then
  - $f_1 \land f_2$, $f_1 \lor f_2$, $\neg f_1$,
  - $AG(f_1)$, $AF(f_1)$, $AX(f_1)$, $EG(f_1)$, $EF(f_1)$, $EX(f_1)$, $E[f_1Uf_2]$, $A[f_1Uf_2]$, $E[f_1Bf_2]$ and $A[f_1Bf_2]$ are CTL formulas.

A CTL formula is always evaluated in a state, a vertex in the computation tree, and we write $q \models F$ to denote the fact that formula $F$ holds in state $q$, or in other words $F$ holds for the computation tree with $q$ as root. The semantics of the CTL operators $AG$, $AF$, $EG$, $EF$ and $EX$ can be described as follows. These operators form a subset of CTL that is sufficient for the purposes of the thesis. The others are not needed, and will therefore not be further described here.

In a computation tree rooted at $q$, we say that
$q = AG(F)$  \iff  $F$ holds in all states of the tree,
$q = AF(F)$  \iff  $F$ holds in some state,
$q = EG(F)$  \iff  $F$ holds in some path from $q$, $F$ is true in all states,
$q = EF(F)$  \iff  $F$ holds in some path from $q$, $F$ is true in some state,
$q = EX(F)$  \iff  $F$ holds for some immediate successor state of $q$.

The following table gives an intuitive interpretation of the CTL operators.

<table>
<thead>
<tr>
<th>operator</th>
<th>Intuition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AG(F)$</td>
<td>$F$ will hold forever.</td>
</tr>
<tr>
<td>$AF(F)$</td>
<td>Eventually, $F$ will hold.</td>
</tr>
<tr>
<td>$EG(F)$</td>
<td>It is possible that $F$ holds forever.</td>
</tr>
<tr>
<td>$EF(F)$</td>
<td>$F$ may eventually hold.</td>
</tr>
<tr>
<td>$EX(F)$</td>
<td>$F$ might hold in the next state.</td>
</tr>
</tbody>
</table>

**Example 6.6** CTL formulae can be used to check various properties of a model. Two important properties are liveness, the fact that something good will eventually happen, and safety, that no undesired things happen. Consider the properties $Good$ and $Bad$. $Good$ holds in all states that perform a task that is important for the performance of the system, if permanently disabled the system is considered malfunctioning. The predicate $Bad$ holds in all states that are undesired, if reached the system is malfunctioning.

A safety property of the system is that states $q$ such that $q \models Bad$ are unreachable. That can be checked by evaluating the following CTL formula.

$$AG(\neg Bad)$$

or, equivalently

$$\neg EF(Bad)$$

That is, there is no state in the computation tree that is $Bad$.

Liveness can be expressed in the following way. There should not be any infinite path of execution that does not include some state $q$ such that $q \models Good$. In other words, in all paths of execution, $Good$ should hold infinitely often. This can be checked by the following CTL formula.

$$AG(AF(Good))$$

In words, it is true for all states ($AG$), that eventually ($AF$) $Good$ will hold. This excludes any infinite computation not containing $Good$ states.

### 6.2.2 The SMV modeling language

The language accepted by SMV [7] describes a set of intercommunicating processes. A process is defined by a set of state variables and a transition relation.

A major difference between the SMV modeling language and the language presented in this chapter is that in SMV there is no support for handshaking
because there is no built-in support for acknowledging the reception of a signal from one process to the other. Therefore, our compiler has to add states in communicating processes for a handshaking protocol if the objects of the system are modeled as individual modules. The whole system could be modeled as one module, with one state variable for each object. Then a handshake protocol would not be needed, and that would probably be a more efficient design. See Section 6.3 for more discussion about computational efficiency of SAD.

6.2.3 Compilation to CTL

In section 5.3 we defined two predicates, present(a) characterizing traces containing a and isroot(a) characterizing traces where a is the first critical event.

In the following, C is a set of critical events, SD is a system description and OBS is an observation.

In section 5.4 we defined the set Q(SD) as the set of global system states. A state \( q \in Q(SD) \) does not in general contain information about which transitions have been used to reach it, but now when we will reason about sequences of events we need that information. This was not needed in section 5.5 since the very same events are enabled whenever the system is in state \( q \) regardless of the previous event history, but now we need the history information because the log of events depends on the history of events. Therefore, in the compilation to CTL we use \( Q' = (Q \times 2^f) \) as the set of system states, and for each state we define the predicate seen where \( (q, A) \models seen(a) \) iff \( a \in A \). The corresponding transition relation on \( Q' \), \( \Rightarrow' \), is defined by

\[
(q_1, A_1) \Rightarrow' (q_2, A_2) \text{ iff } q_1 \xrightarrow{a,b} q_2 \text{ and } A_2 = A_1 \cup \{a, b\}
\]

Example 6.7 Assume that we have a system described by the following code.

class Simple
    s1 - a! -> s2 - b! -> s1
instances
    s: Simple

Now \( Q = \{s.s1, s.s2\} \), but \( Q' = \{(s.s1, \emptyset), (s.s1, \{s.a!\}), \ldots\} \). The relevant part of the transition relation, i.e. the part reachable from the initial state \( (s.s1, \emptyset) \), is defined by the following.

\[
\begin{align*}
(s.s1, \emptyset) & \Rightarrow' (s.s2, \{s.a!\}) \\
(s.s2, \{s.a!\}) & \Rightarrow' (s.s1, \{s.a!, s.b!\}) \\
(s.s1, \{s.a!, s.b!\}) & \Rightarrow' (s.s2, \{s.a!, s.b!\}) \\
(s.s2, \{s.a!, s.b!\}) & \Rightarrow' (s.s1, \{s.a!, s.b!\})
\end{align*}
\]

To simplify the presentation, we define the predicate \( \text{obsOK} \subseteq Q' \) as

\[
\text{obsOK} \equiv \forall a \in OBS, \text{seen}(a) \land EG(\forall a \in (\mathcal{E}_{obs} - OBS), \neg \text{seen}(a))
\]
6.2. SMV REPRESENTATION OF SYSTEM DESCRIPTIONS

We say that an event \( a \) is an enabled root if \( \{ SD, OBS, isroot(a) \} \) is consistent. This translates to the following CTL formula where \( a \) is universally quantified and \( crit(a) \) holds.

\[
\forall a \in crit, EF(\neg\text{seen}(a) \land EX(\text{seen}(a) \land EF(\text{obsOK})))
\]  \hfill (6.1)

In words, an event \( a \) 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 \( s_2 \),
- event \( a \) occurs in state \( s_2 \),
- a state \( s_3 \), where all observed events have occurred, is reachable from \( s_2 \).

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

\[
AG(\text{obsOK} \rightarrow AF(\text{seen}(a) \lor \neg\text{obsOK}))
\]  \hfill (6.2)

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

**Example 6.8** Assume that we have the following input file to the compiler.

class:Client
  idle - call! -> wait - return? -> idle

class:Server
  idle - call? -> computing - return! -> idle

instances
  c : Client
  s : Server

connects
  c <-> s

observations
  s.return!

critical
  c.call!, s.return!

The resulting SMV code for the Server is shown in Figure 6.1. The first line of code defines the input signals to the SMV module that represents the Server.
class. The first signal corresponds to receiving a call-event from some other object and the second signal is the acknowledgment signal used when sending a return-signal. We model the system as a set of running objects, competing for a limited resource of computing power. The variable running defined on line 3 is used to indicate if the object has access to the processor, or is idle waiting.

On lines 4 and 5, the states of the object are defined. The states are the states from the input model and then three derived states needed for the handshaking protocol. The names of the derived states begin with underscore. The state _call_acking_computing is intended to be read "acknowledging call, in next step go to state computing". The state _return-ing_idle has the intuitive interpretation "trying to send return, then go to state idle", whereas the state _w_ack_return_idle means "wait for acknowledgment of return, then go to state idle".

Lines 7 and 8 take care of the running variable. Initially it is 0, and in any step thereafter it is chosen to be either 0 or 1. In line 9 the initial state of the object is set to idle.

Disregarding the handshaking protocol for inter-object event transmission, the object state would be unchanged if running is zero, and otherwise it would follow the transitions in the definition. This is reflected in lines 17 to 19. Line 17 determines that the next state will be the same as the present if running is zero. Line 12 states that if the state is idle and we have a call-signal from the environment, we change state to computing. From computing we go to idle, sending return, and this is reflected by line 18. Line 19 determines that if nothing above applies, the object state is unchanged.

The handshake protocol is defined by lines 13-16 and 22-24. The protocol has priority over the running variable, since the events taken by the protocol have to be synchronized between two object and although it takes several ticks in the SMV model, the intended understanding of the procedure is that it is atomic and therefore not interrupted by processor scheduling, i.e. running going from 1 to 0.

Lines 13 and 14 are needed to have a state that can be associated with the sending of the _ack_call and return-events respectively, which is done in line 22 and 23. When waiting for acknowledgment of the return-event, lines 15-16 chose the next state depending on whether the acknowledgment is present or not.

Line 24 defines the _sent_return-signal that states that the object has sent the return-signal and received an acknowledgment. If acknowledgment is not received, the event is not regarded as sent.

Due to the simplicity of the components in this toy example, the SMV code for the client is very similar. Two more modules are needed in SMV to be able to reason about the system. First, the predicate seen described earlier, holds for a state and an event if the event is part of the trace leading to the state. Such a property is not automatically expressible in SMV, where only momentary state can be used in the CTL formulae. Therefore the module _Indicator is used to implement a simple event observer that detects and remembers the presence of a relevant event. Second, the module main is used in SMV to instantiate
6.2. SMV Representation of System Descriptions

1 MODULE Server(call_in, _ack_return_in)
2 VAR
3 running : boolean;
4 state : {idle, _call_acking_computing, computing, _return-ing_idle,
5 _w_ack_return_idle};
6 ASSIGN
7 init(running) := 0;
8 next(running) := {0, 1};
9 init(state) := idle;
10 next(state) :=
11 case
12 state = idle & call_in : _call_acking_computing;
13 state = _call_acking_computing : computing;
14 state = _return-ing_idle : _w_ack_return_idle;
15 state = _w_ack_return_idle & _ack_return_in : idle;
16 state = _w_ack_return_idle & !_ack_return_in : computing;
17 !running : state;
18 state = computing : _return-ing_idle;
19 1 : state;
20 esac;
21 DEFINE
22 return := state = _return-ing_idle;
23 _ack_call := state = _call_acking_computing;
24 _sent_return := state = _w_ack_return_idle & _ack_return_in;
25 FAIRNESS
26 running

Figure 6.1: The SMV code for the example Server class.

processes from the given modules. There will be one instance for each object, and one observer instance for each signal that is used in the following SPEC commands. The two modules are defined as in Figure 6.2.

Now, we are ready to actually use the model and evaluate CTL formulae. This is done with the SPEC command in SMV. The following code is used to test if any of the two critical events are enabled roots, see CTL formula 6.1 for the general formula for this property.

SPEC EF(!(c_out_call.seen) & EX(s_out_return.seen &
  EF(s_out_return.seen)))
SPEC EF(!(s_out_return.seen) & EX(c_out_call.seen &
  EF(s_out_return.seen)))

Now, since SMV evaluates the first formula to false and the second to true, the compiler concludes that there is only one enabled root among the critical events and is thus done.
CHAPTER 6. A PROTOTYPE IMPLEMENTATION

MODULE _Indicator(event)
VAR
    seen: boolean;
ASSIGN
    init(seen) := 0;
    next(seen) := case
        event : 1;
        1 : seen;
    esac;

MODULE main
VAR
    c : Client(s._ack_call, s.return);
    s : Server(c.call, c._ack_return);
    s_out_return : _Indicator(s._sent_return);
    c_out_call : _Indicator(c._sent_call);

Figure 6.2: The SMV code for the event observer and main module.

If the compiler has a remaining set of more than one event after this first phase of SPECs, it tests for present events among the enabled roots. This is done by SMV code similar to the following, which is a special case of the general CTL Formula 6.2.

SPEC AG(s_out_return.seen -> AF(c._sent_call))

In this case the observations do not contain negation, and thus obsOK is monotone in the sense that once it holds, it will hold for ever. No matter what happens the observed events will remain seen. Thus, the CTL formula 6.2 can in this case be simplified.

6.3 Computational Efficiency

As mentioned above, computational efficiency has not been a primary concern of this prototype design. The prototype implementation of SAD has been used to find out what can be expressed in temporal logic and computed when it comes to causal dependencies reasoning, leaving to future research the question of how it can be computed efficiently.

There are several possible improvements that could be tried for improvement of the computational efficiency. Encoding the whole system model into one SMV module with state transitions according to $Q'$ as described in Section 6.2.3 would remove the need for explicit encoding of the handshake protocol. This improvement has great potential, but probably an even more efficient way of improving the execution efficiency is to change to an event-based CTL model-checker instead of SMV that is state-based. The CTL formulas that are evaluated by
SAD are rather complex, but do not show much variation. Therefore, a tailored system using model-checking techniques tuned to perform the very queries that SAD needs is probably the most efficient solution.

6.4 SAD Execution Examples

SAD 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 left.

The phases are consistency check, enabled roots and present roots in that order. Each phase corresponds to one line of output from SAD. The line is a heading identifying the phase followed by two sequences of events, the second one within parenthesis. The first sequence represents the events remaining in the set of suggested root events and the second sequence represents the set of events that have been removed from the set during the phase.

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, so the first line of output will always end with an empty pair of parenthesis. 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.

6.4.1 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 and observations. If we do not find a proven root, we are in one of three possible situations described in the following sections.

Example 6.9 In this example we have three objects a, b and c, instances of the same class. Illustrated in Figure 6.3, the class describes a process that can change between behaving as a client and as a server. When in the state 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. Messages regarding down and fail are considered critical, i.e. possible causes of system failures.

In this example 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 the 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 only be two fail-transmissions and thus a.fail! was the first.

The scenario description is as follows.

class: Serent
  idle - call! -> wait - return? -> idle
  wait - fail? -> err - down! -> err
  idle - down? -> stop
  idle - call? -> processing - return! -> idle
  processing - fail! -> idle

instances
  a, b, c : Serent

connects
  a<-b, b<-c, a<-c

observations
  a.fail?, a.fail!

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

In the second phase SAD finds that there is only one enabled root event.

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

6.4.2 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 6.10 Here we use the same system as described in the example in
section 6.9, but we let the observations be different. The observation a.fail?,
!c.fail! means that we know that object a has received a fail-message, and
we know that object c has not sent it. 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 can still be fulfilled 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 scenario description is as follows. Here, and in the following input
examples, we have omitted all sections that are identical in all the examples,
leaving only the observations section.

observations
  a.fail?, !c.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
  c.fail!, a.fail!, b.fail!, c.down!, b.down!, a.down! ()
In the second phase, the set of candidates is reduced to two events.

Enabled roots
\[ a.f \!\! \text{ail}, \ b.f \!\! \text{ail} (c.f \!\! \text{ail}, \ c.d \!\! \text{own}, \ b.d \!\! \text{own}, \ a.d \!\! \text{own}) \]

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

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

### 6.4.3 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 6.11** Here we use the same system as described in the example in section 6.9, but we let the observation 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 \( a.f \!\! \text{ail}, \ b.f \!\! \text{ail} \) or \( c.f \!\! \text{ail} \) are root events. None of them are present, though. Therefore, we do not have any strong root candidates in this scenario.

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

**Observations**
\[ a.f \!\! \text{ail} \]

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 events are present.

**After consistency check**
\[ c.d \!\! \text{own}, a.f \!\! \text{ail}, a.d \!\! \text{own}, b.f \!\! \text{ail}, b.d \!\! \text{own}, c.f \!\! \text{ail} () \]

Enabled roots
\[ a.f \!\! \text{ail}, b.f \!\! \text{ail}, c.f \!\! \text{ail} (c.d \!\! \text{own}, a.d \!\! \text{own}, b.d \!\! \text{own}) \]

Present roots
\[ a.f \!\! \text{ail}, b.f \!\! \text{ail}, c.f \!\! \text{ail} () \]

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.
6.4. SAD EXECUTION EXAMPLES

6.4.4 Several Strong Root Candidates

If we have 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 6.12** Again, we use the same system as described in the example in section 6.9, 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.

**Observations**

b.fail!, c.fail!

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**

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

**Present roots**

b.fail!, c.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 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.
Chapter 7

Conclusions and Future Work

We have presented an approach to fault isolation in object-oriented control systems using a behavioral system model. We have provided a formal framework for post-mortem reasoning about behavior and causal dependencies between messages from the system, as well as a prototype implementation using a model-checker for behavioral reasoning.

It has been the aim from the beginning of this work to integrate the results with previous work, i.e. DrRobot - see [9, 8, 10]. The natural way of performing such integration is to use SAD, described in this thesis, as a second phase of fault isolation after a first stage of DrRobot processing. In that way, when DrRobot finds a unique fault, the second phase will not be used, but in the cases when DrRobot fails to come up with a unique fault or no fault at all, the second phase will be tried.

Since the system model used in DrRobot is static and structural, the dependencies between classes in the system are the same no matter what we know about the present scenario. A relationship between two classes in the class diagram determines that the classes exhibit a dependency in some scenario, and is therefore used in all scenarios. This problem does not only make the fault isolation coarse, it can also make it impossible in less benign cases. Sometimes the structural model, a model of all possible executions, contains cyclic dependencies that are never present in any scenario. In this case the structural fault isolation cannot infer anything about the dependencies between the objects in the cycle without resorting to ad hoc methods.

When having a second stage of analysis, the DrRobot part of the fault isolation system will not have to resort to ad hoc methods in selecting which explanations to follow when the structural model demands it. Instead, it leaves the decisions to the SAD phase where a behavioral system model enables discrimination between dependencies that are present only in some scenarios.

It will be a topic of future work to find design principles for constructing systems that allow for successful fault isolation. Even with a behavioral system model some scenarios may be impossible to distinguish from each other thus fault isolation may fail, see e.g. Sampath et al [19]. Some events that are relevant
to the fault isolation might not be reported, or are perhaps not even observable.

These situations could be detected at design time, and the designer would then have the opportunity to alter the design in such a way that the scenarios become distinguishable. One way of doing this is to add observer objects, or more liberal error reporting functionality in present objects, that have the responsibility of monitoring the state of the subsystem that is not observed well enough. Having such design support, it will become possible to reason formally about the quality of the fault isolation and prove correctness.

Another area of future research is improvement of computational efficiency. This will be especially needed if the approach is used as a design support tool, making response time a more important factor than in the case of post mortem fault isolation. In Section 6.3 we gave examples of such future improvements.
Bibliography


Vojin Plavsic: Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)


Ola Strömors: A Structure Editor for Documents and Programs, 1986.


Peter Åberg: Design of a Multiple View Presentation and Interaction Manager, 1989.

Henrik Eriksson: A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.


Johan Ringström: Compiler Generation for Parallel Languages from Denotational Specifications, 1993.


Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag. 1996.
High-Level Synthesis under Local Timing Constraints, 1996.
Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
Multidatabase Integration using Polymorphic Queries and Views, 1996.
The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
Compilation of Mathematical Models to Parallel Code, 1996.
Exploration of Polygonal Environments, 1996.
Simulation and Data Collection in Battle Training, 1996.
Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
Chefers användning av kommunikationsteknik, 1996.
Staffan Flodin: Efficient Management of Object-Oriented Queries with Late Binding, 1996.
Magnus Werner: Multidatabase Integration using Polymorphic Queries and Views, 1996.
Mikael Lind: Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
Kristina Larsen: Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag, 1996.
Niclas Andersson: Compilation of Mathematical Models to Parallel Code, 1996.
Jörgen Lindström: Chefers användning av kommunikationsteknik, 1996.
| No 769 | Pawel Pietrzak: | Integrating C++ memory management with Stroustrup's RAII - a modular and efficient design, 1999. |

---

| No 613 | Jens-Olof Lindh: | Analys och värdering av systemutvecklingsmodeller och dess användning, 1997. |
| No 587 | Anders Bäckström: | Integrating C++ memory management with Stroustrup's RAII - a modular and efficient design, 1999. |
| No 562 | Fredrik Eklund: | Declarative Error Diagnosis of GAPLog Programs, 1997. |

---


---

| No 742 | Pawel Pietrzak: | Static Incorrectness Diagnosis of CLP (FD), 1999. |
| No 748 | Tobias Ritzau: | Integrating C++ memory management with Stroustrup's RAII - a modular and efficient design, 1999. |
| No 753 | Johanna Eklund: | Integrating C++ memory management with Stroustrup's RAII - a modular and efficient design, 1999. |
| No 760 | Mats Gustafsson: | Bringing Role-Based Access Control to Distributed Systems, 1997. |

---

| No 768 | Per-Ove Zetterlund: | Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998. |
| No 776 | Martin V. Howard: | Designing dynamic visualizations of temporal data, 1999. |

---

<p>| No 788 | Svein Bergum: | Managerial communication in telework, 2000. |</p>
<table>
<thead>
<tr>
<th>No</th>
<th>Name</th>
<th>Title</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>823</td>
<td>Jean Paul Meynard</td>
<td>Control of industrial robots through high-level task programming, 2000.</td>
<td></td>
</tr>
<tr>
<td>842</td>
<td>Magnus Kald</td>
<td>The role of management control systems in strategic business units, 2000.</td>
<td></td>
</tr>
</tbody>
</table>
Towards Behavioral Model Fault Isolation for Object Oriented Control Systems

Dan Lawesson

We use a system model expressed in a subset of the Unified Modeling Language to perform fault isolation in large object oriented control systems. Due to the severity of the failures considered and the safety critical nature of the system we cannot perform fault isolation on-line. Thus, we perform post mortem fault isolation which has implications in terms of the information available; the temporal order in the error log can not be trusted. In our previous work we have used a structural model for fault isolation. In this thesis we provide a formal framework and a prototype implementation of an approach taking benefit of a behavioral model. This gives opportunities to perform more sophisticated reasoning at the cost of a more detailed system model. We use a model-checker to reason about causal dependencies among the events of the modeled system. The model-checker performs reasoning about temporal dependencies among the events in the system model and the scenario at hand, allowing for conclusions about the causal relation between the events of the scenario. This knowledge can then be transferred to the corresponding fault in the system, allowing us to pinpoint the cause of a system failure among a set of potential causes.

default isolation, model-checking, UML, object orientation