Linköping Studies in Science and Technology. Dissertations
No. 608

Behavioral and Structural Model Based Approaches to Discrete Diagnosis

Magnus Larsson

Department of Electrical Engineering
Linköping University, SE–581 83 Linköping, Sweden

Linköping 1999

Behavioral and Structural Model Based Approaches to Discrete Diagnosis

`magnusl@isy.liu.se`
`http://control.isy.liu.se`
Department of Electrical Engineering
Linköping University
SE–581 83 Linköping
Sweden

ISBN 91-7219-616-5 ISSN 0345-7524

Printed in Sweden by Linus o Linnéa AB

To Ann

Abstract

The basic motivation for this thesis is the fact that things go wrong. With the growing complexity of todays engineering systems, the need has arisen for systematic approaches to failure diagnosis, i.e., fault detection and isolation.

In the first part of this thesis an approach for modeling and diagnosis of systems that fall in the area of discrete event dynamic systems is proposed. The approach is applicable to systems that at some level of abstraction have an interesting discrete event dynamics that can display faulty behavior. The systems suitable for this approach typically consist of several interacting components where abrupt, but non-catastrophic, faults can occur in the components.

We use a relational framework for discrete event dynamic systems focusing on a conceptually simple representation of the relationship between inputs, outputs and states of a discrete event system. Faults and faulty behavior are modeled locally using the state variables, and the diagnosis problem basically is to infer the possible states of the system using the system model and observations of the real system, i.e., an observer problem. Detectability and isolatability properties are defined and algorithms for analysis are proposed. The transitions necessary and sufficient for detection can automatically be computed from the system model under certain conditions. We also show how to compute the finest possible fault partition.

The second part of this thesis addresses the problem of fault propagation between software modules in a large-scale control system with object oriented architecture. There exists a conflict between object-oriented design goals such as encapsulation and modularity, and the possibility to suppress propagating error conditions. When an object detects an error condition, it is not desirable to perform the extensive querying of other objects that would be necessary to decide how close to the real fault the object is and hence whether it should report to the user.

The fault propagation manifests itself as many irrelevant error messages and hence causes problems for system operators and service personnel trying to quickly isolate the real fault. A system developer with insight in the internal system design, can, of course, often easily interpret the multitude of error messages from a fault scenario and isolate the root cause. The key observation is that it can often be done using mental high-level models of the system and the mechanics of the fault propagation. We have made an effort to automate this procedure, and propose a fault isolation scheme as an extra layer between the operator and the core control system. In the fault isolation layer, post-processing of the fault information from the system is performed, to achieve clear and concise fault information to the operator without violating encapsulation and modularity.

A high-level and informal explanation model for the fault propagation is presented and a taxonomy for error conditions in an object oriented system is proposed. We present algorithms and methods that use the explanation model and the error condition taxonomy together with a structural system model to form a cause-effect relation on the error messages, that can be used to find the most significant error message(s) in a fault scenario.

Acknowledgments

First of all I would like to thank my supervisor Prof. Lennart Ljung for giving me the opportunity to join the Automatic Control group and for creating the truly inspiring and friendly atmosphere that surrounds the group. I would also like to thank Lennart and my co-supervisor Inger Klein for the encouragement, guidance and many insightful discussions.

The project presented in the second part of this thesis was carried out in close cooperation with ABB Robotics in Västerås and the people there. I am grateful to Torgny Brogårdh, who originally initiated the project and gave me this opportunity. I would also especially like to thank Peter Ericsson, whose knowledge and dedication I have greatly benefitted from. Many others of the very nice and dedicated people at ABB Robotics have happily answered my questions and otherwise helped me during this work, and they are hereby gratefully acknowledged.

I have enjoyed many fruitful discussions and joint work with Dan Lawesson and Ulf Nilsson. Dan and Inger have also implemented parts of the fault isolation tool *DrRobot*. Anders Henriksson and I have had several interesting discussions about model based diagnosis in AI.

Lennart, Inger, Krister Edström, Dan, Ulf, Måns Östring and Mikael Norrlöf have carefully read the whole or parts of the manuscript and their comments has significantly improved the final result. For this I am very grateful. Anything still wrong or unclear is my fault alone.

I would like to thank Niclas Bergman for help with LaTeX, Anders Stenman for help with XEmacs and Mattias Olofsson for keeping the computers happy.

The Control group in Linköping is a great place to be, and all members of the group are hereby gratefully acknowledged for being such a nice and helpful bunch. Someone who deserves some extra attention is our secretary Ulla Salaneck, without whom the group would come to a grinding halt before anyone could say ”who knows that?”.

The project presented in Part II was supported by ABB Robotics within the NUTEK Competence Center ISIS (Information Systems for Industrial Control and Supervision), which is gratefully acknowledged.

My friends and loved ones deserve many deeply felt thanks for distracting me and helping me get my mind of work. My sister Åsa, her Urban and children Rebecka and Alexander have provided me with a nearby refuge with a warm atmosphere and where you never need to go unoccupied. I would like to thank my parents, Ulla and Lennart, for all the love and support they have given me always. Finally, I would like to thank Ann for the understanding, love and support during the writing of this thesis.

Linköping, October 1999

Magnus Larsson

Contents

Chapter 1

Introduction

The basic motivation for this thesis is that things go wrong. More specifically, in engineering systems things stop working or break, a valve gets stuck, a communication link breaks down, an engine gets overheated, etc. These faults need to be timely and accurately detected and isolated for safety, reliability and availability of the system.

If a fault is detected late, it might spread and cause unnecessary operation disturbances or even material or personal damage. Also, a fault may well be easy to fix, but hard to find. A skilled operator or reparateur can often quickly isolate the fault in a system he is familiar with, but as systems grow more complex this 'manual' diagnosis becomes more difficult, and the specialists capable of performing it become fewer and more expensive to train. In industry today, complex systems are often bought 'off the shelf' and it takes time to build up the necessary competence to do even rudimentary fault diagnosis. It is therefore of interest both to customer and vendor to have, if not totally automated, so at least more sophisticated diagnosis tools to their complex systems. Even if the diagnosis tools only can handle quite simple faults, they might be of great value.

The focus here is on man-made systems, but the concept of diagnosis is still very broad and depends on the nature of the faults and the type of system under consideration. Are we for instance interested in monitoring whether some equipment is working properly, and noisy measurements and complex continuous dynamics are the main problems? Or do we need to interpret a large amount of error messages and alarms generated by the system to find the one underlying cause?

Due to the wide flora of diagnosis problems of fundamentally varying nature, we believe that there is no such thing as a universal solution or framework for diagnosis, and that the approach used for a certain diagnosis problem or class of problems must be carefully chosen and tuned for that specific purpose. In this thesis we will with the term diagnosis mean fault detection and fault isolation, i.e., from observations of the system determine if a fault has occured, where it occured and also which specific fault is was.

The thesis consists of two parts, with diagnosis approaches for two classes of systems.

In Part I, an approach for modeling and diagnosis of systems that fall in the area of discrete event dynamic systems (DEDS or DES) is proposed. The approach is applicable to all systems that at some level of abstraction have an interesting discrete event dynamics that can display faulty behavior. The systems suitable for this approach typically consist of several interacting components where abrupt, but non-catastrophic, faults can occur in the components. Some close connections to other approaches in the literature will also be discussed.

Part II of the thesis develops a high level fault isolation method suitable for object oriented control systems. Knowledge of the system structure is used to discriminate among a stream of error messages without breaking the object oriented design goals of encapsulation and modularity. The models and extra information needed for the isolation is a natural part of the normal software process.

The problem of diagnosis has been widely addressed in the literature and an overview of some approaches is made in Section 1.1. The main contributions of this work are summarized in Section 1.2 and an outline of the rest of the thesis is given in Section 1.3.

1.1 Overview of diagnosis approaches

A traditional approach to diagnosis used in many industrial systems is simple limit checking of signal values and predefined threshold logic. For example, if a sensor value leaves its normal range, an alarm is generated. Due to failure propagation in the system, often a number of alarms are set off and then the problem is to isolate the basic cause. Common methods used are fault trees and expert systems, even though these methods of course not are restricted to alarm analysis only. Section 1.1.1 gives a brief overview of traditional methods like fault trees and expert systems and also some other interesting approaches to diagnosis.

The concept of using a model of the system to aid the diagnosis has been in the literature at least since the 70's (see, e.g., [80]) and has turned into several very active research fields, see Section 1.1.2.

1.1.1 Traditional and non-model based approaches

In this section we briefly describe some approaches with the common feature that they explicitly associate an observed behavior with a cause, like a fault. The kind of pattern-matching "models" these approaches use are often called *associative*, or shallow.

Fault trees

Fault tree analysis is a widely used technique in the process control industry for reliability analysis, fault detection and isolation. The basic idea is that a failure can trigger other failures or events in the system and hence be traced back to the basic cause [34, 77]. A fault tree graphically represents a cause-effect relationship in the system. The root of a fault tree, the so called TOP event, can for example be a system failure. A tree with the possibly contributing events or basic faults as leaves, and inner nodes of AND- and OR-type, is then constructed, often manually from the system specification. Sets of events that trigger the top event (*cut sets* and *minimal cut sets*) are computed. By assigning probabilities to the basic events, a failure probability can be found.

Manual fault tree construction is laborious and error prone, and much work has been done on computer assisted and automatic fault tree construction, see for instance [41, 23, 78].

Expert systems

A popular method for diagnosis and supervision of complex systems has been the use of expert systems, often in conjuction with fault tree structures, see, e.g., [72, 76]. Expert systems are especially well suited for systems that are difficult to model, with complex interactions between and within components. Domain experts have heuristic knowledge of the system and of how symptoms relate to faults. In traditional expert systems, this knowledge is represented in a rule base and used in conjunction with an inference engine.

This heuristic approach has several drawbacks. Acquiring knowledge from experts is difficult and time consuming, and for new systems a considerable amount of time might elapse before enough knowledge has accumulated to make reliable diagnosis possible. The rule base is very domain dependent and not reusable, and the maintenance of large rule bases is difficult.

Chronicles

An expert system-like approach to diagnosis and supervision of dynamic systems using temporal information, is the real-time situation recognition method described in [13, 51]. For the system under consideration, a set of events, obtainable for instance by signal processing, is defined. A number of situations that are considered desirable to recognize are characterized with these events and temporal constraints among the events. Such a characterization of a situation is called a *chronicle* and can correspond both to correct operation and failures of the system. A situation recognizer is then fed with observed and time-stamped events in real-time and can notify an operator for example when a failure situation has occured or is under development. Of course only known situations can be recognized. See also the concluding comments on expert systems above.

A state based DES approach to active diagnosis

A discrete event system approach for active diagnosis is proposed by Lin [47] and further treated in [3]. The system is modeled as a finite state machine, where the states of the machine describe conditions of the components. The only dynamics in the FSM is that the system can go from normal to faulty, and no normal behavior is modeled. The observations (sensor readings) are included as a mapping from the fault states to certain observable events (tests). To diagnose a fault is to identify which state or set of states the system belongs to. Off-line and on-line diagnosis are treated separately.

In off-line diagnosis the system is thought to be in a test-bed, where the system does not change state unless forced to as part of the diagnosis. Tests can be performed, and the outcome of the tests are called observable events. The off-line diagnosability of the system is analyzed with respect to a fault partition and the set of observable events (i.e., tests). Since the system is in a test-bed, the order in which the tests are performed does not affect the diagnosability.

In on-line diagnosis, the system is in normal operation, and hence it can change state uncontrollably. The system is assumed to be partial state observable via an output map. Given a state (fault) partition, the system is defined to be on-line diagnosable if there exists a sequence of commands so that the state of the system, up to the fault partition, can be decided from the output map. An algorithm for computing such a sequence is given in [47].

1.1.2 Model based approaches

The basic idea in all model based diagnosis is to compare observations of the real system with the predictions from a model. In the case of a fault, a discrepancy between the actual, observed behavior and the predicted, normal behavior arise. This discrepancy can then be used to detect, isolate and identify the fault depending on the type of model and methods used.

The kind of models used by these approaches are often called *explanatory*, or deep models. Several approaches to model based diagnosis with model paradigms ranging from differential equations to qualitative behavior models have appeared in the literature and been proven useful in practice.

Analytical redundancy methods

In the control system community, the most common class of model based diagnosis methods proposed, and the most successful one, are the analytical redundancy methods, see for instance [80, 58, 24, 54, 67] and references therein. The method is based on the fact that observed signals from the system, such as sensor measurements and control signals, contain information of the system state. The desired state information can be extracted with the help of a good differential equation model of the system, often obtained by physical modeling, or system identification [48].

In general, these methods consist of two steps. First residuals are generated by comparing observed signals from the system with predicted values. These residuals are usually designed to be zero if no fault is present. Second, a fault detection and isolation step is employed, using these residuals as input. One of the main problems with this approach is the difficulty in acquiring good enough models. The demands on the accuracy of the models are usually higher than for control design, since the residual generator works in open-loop. Robust methods for residual generation has received considerable attention in recent years and is an active research area, see, e.g., [58, 59].

Recently, there has also been some efforts towards formally demonstrating the connections between the analytical redundancy methods and the model-based approaches in AI briefly described in the next section [54]. The basically set-theoretic but variable-based description used, is similar to the relational approach presented in Chapter 3 in this thesis.

Model based approaches in AI

In artifical intelligence, model-based diagnosis, or diagnosis from first principles, is pioneered by Reiter [65]. The basic idea is to predict the behavior of the system using behavioral and structural models of the system and its components and compare it with observations of the actual behavior of the real system. Two main characterizations of model based diagnosis exists in the literature, consistency based diagnosis [65] and abductive diagnosis [61]. The former needs a model of the correct behavior only, and a diagnosis is a set of components that when assumed non-correct, makes the predicted behavior consistent with the observed. The abductive approach must have models of the faulty behavior, that then has to be known. A diagnosis is a set of component faults that is not only consistent with, but also explicitly predicts the observed behavior. A good introduction to model-based diagnosis is Chapter 2 of [26], and for a comprehensive collection of literature, see [32].

Model based diagnosis in AI is mainly aimed at static systems and especially at troubleshooting digital circuits. The extension of the methodology to dynamic and time-varying systems remains an active research area, see, e.g., [26, 15, 16, 79].

Livingstone

An AI model based approach that deserves special mentioning is based on Livingstone, that, in the authors' own words, is an implemented kernel for a model-based reactive self-configuring autonomous system. See further Williams and Nayak [79]. Livingstone is part of the system architecture in DS1, an autonomous spacecraft project, part of NASA's New Millennium Program [53]. It performs diagnosis and recommends reconfiguration actions to recover from failures.

Livingstone uses a model of the components of a complex system, which is very similar to the DEDS models presented in Chapter 5 in this thesis. The difference is mainly in the framework and the representation. The local modeling approach for flow/pressure systems in Section 5.8, was partly inspired by Livingstone.

The formal language approach

Sampath et al. has proposed a formal language framework for diagnosis and analysis of diagnosability properties of un-timed discrete event systems [70, 71, 68]. The approach is closely related to the Ramadge-Wonham framework for supervisional control of DES [63]. The method has mainly been applied to HVAC-systems (heating, ventilation and air-condition), but is applicable to all systems that at some level can be meaningfully modeled as discrete event systems. The main features of the approach is methods for modeling the normal and faulty behavior of systems as DES, implementation of online passive diagnosis and analysis of diagnosability properties.

Both the normal and faulty behavior of the system to be diagnosed is modeled with a finite state machine. Components are modeled locally and faults in the components are modeled as unobservable events. The system model is then put together with the usual synchronous composition [63]. To include sensor information in the model, global sensor maps from the states of the model to sensor readings are constructed and then included in the event labels.

The diagnostic problem is then to infer about past occurences of unobservable fault events from the observable events. A diagnoser that gives a state estimation of the system after each event is constructed from the model. The diagnoser also maintains information about the possibility of fault event occurences using so called fault labels attached to the state estimates. The diagnoser hence basically is an extended observer.

One of the main features of this approach is the ability to analyze diagnosability properties. Diagnosability is, loosely put, the ability to detect and isolate an occured fault with finite delay during normal operation of the system using the observable events. Isolation is defined w.r.t. a fault partition.

To demand that a fault should always be diagnosed with finite delay is rather strong, and very few systems can fulfill that. Therefore the notion of I-diagnosability is introduced, which means that a fault should be detected and, up to a given partition, isolated with finite delay *after* a so called indicator event has occured. Put in other words, the system has to be excited enough for the fault to be diagnosed.

If a model of a system is not diagnosable, or I-diagnosable, the authors identify two means of making it diagnosable: i) introduce more sensors and ii) design the controller so that the faulty behavior is excited and can be detected. The theory is set in a formal language framework and is very closely related to the Ramadge-Wonham (RW) framework for supervisory control of discrete event systems [63]. In [69] and [68], the authors present a method for designing controllers in the RW-framework that make the system diagnosable.

Finally, we mention that Chen and Provan has extended the approach to timed discrete event systems [14].

Templates

A method for fault monitoring of automated manufacturing systems using the timing and sequencing of events has recently been developed by Holloway et al. [35, 36, 18, 57]. So called time templates or condition templates are used to specify the expected, correct timing and sequencing of events. Templates are designed to be easily implemented on distributed control architectures and are used to detect deviations from the correct behavior. Specific faults cannot be diagnosed. The method is capable of monitoring event sequences corresponding to an arbitrary number of concurrent timed automata, and a timed automata specification can be automatically translated to templates [35, 57]. The manual construction of templates is probably not feasible and work has been done on identifying templates from observations of sequences of events [18].

1.2 Contributions of the thesis

The main contributions of Part I of the thesis are the following:

- A formulation of the diagnosis problem for DEDS in the relational framework.
- A demonstration of the tight connection with model based diagnosis in AI, and of how decision diagram structures can be used to compute diagnoses.
- An effort towards systematic local discrete modeling of physical behavior.
- An automatic procedure that, under certain assumptions, finds which faults are detectable and the necessary behavior for detection.
- An automatic procedure that, under certain assumptions, partitions all faults into sets of indistinguishable faults.

The two latter are not handled in the formal language approach [68], where we can obtain our results by assuming less general models when doing the analysis. Some of the results have been presented at the 37th IEEE Conference on Decision and Control [44].

The main contributions of Part II of the thesis are:

- A general, scalable and easily maintainable scheme to handle fault propagation in large scale control systems with object oriented software architecture.
- A, to the authors knowledge, novel classification of run-time error messages in control systems with object oriented architecture.
- A formal representation of a fault scenario, and extension algorithms using a structural system model.
- An implemented fault isolation tool, useful both for development and testing of the control system, and as fault isolation support for end-users.

ABB Robotics and the author have filed a patent claim for the fault isolation scheme with the swedish patent office. The main content of Part II is also published as a technical report [45].

1.3 Outline of the thesis

Part I Chapter 2 briefly introduces Part I and Chapter 3 introduces the basic concepts of the relational framework and DEDS system theory. The standalone Chapter 4 discusses effective representation and computation with relations using binary and integer decision diagrams. The chapter is primarily intended for readers unfamiliar with BDDs and decision diagrams. DEDS models, modeling and fault modeling in the relational framework are treated in Chapter 5, where also a framework with reusable component models for simple physical flow/pressure modeling is presented. Diagnosis in the relational framework, i.e., fault detection and isolation, is treated in Chapter 7 and analysis of diagnosis properties in Chapter 8.

Part II Chapter 10 introduces Part II and Chapter 11 introduce the basic concepts of object orientated software architecture and the Unified Modeling Language (UML). An industrial robot control system from ABB Robotics is also briefly described. Chapter 12 specifies the structural system model used for fault isolation in Part II. Chapter 13 discusses the assumptions underlying the fault isolation scheme, and introduces a formal representation of a fault scenario. The details of the fault isolation scheme are presented in Chapter 14. The implementation and several examples from the ABB Robotics control system are demonstrated in Chapter 15.

The thesis do not contain a list of the employed notation, but the index is intended to serve the purpose of quickly finding definitions.

Part I

Modeling and Diagnosis of Discrete Event Dynamic Systems

Chapter 2

Introduction Part I

In Part I of this thesis we address basically the same problem as in Sampath et al. [70], i.e., diagnosis of systems with abrupt, but non-catastrophic faults that do not necessarily bring the system to a halt. The approach is applicable to all systems that at some level of abstraction has an interesting discrete event dynamic that can display faulty behavior.

We will set the problem in a slightly different framework than formal languages, namely the relational framework for discrete event dynamic systems introduced by Germundsson [27] and further treated by Gunnarsson and Plantin [39, 31, 60, 30]. This framework is focused on a conceptually simple representation of the relationships between inputs, outputs and states of a discrete event system. Inspired by modern control theory we use variables to represent the piecewise constant signals in a discrete event system. The model is a mapping from the states and inputs to the outputs and next states, represented by a mathematical object called a relation.

The models, and methods for computing diagnoses have a strong connection to the model based diagnosis methods in AI, see the brief review in Section 1.1.2 and Section 7.3. The difference is mainly in how the diagnosis is computed.

The diagnosis approach consists basically of three parts:

i) Modeling of the system to be diagnosed, by constructing a discrete event model containing both the normal and faulty behavior (Chapter 5).

ii) Fault detection and fault isolation using observations from the real system (Chapter 7).

iii) Analysis of diagnosability properties of the system (Chapter 8).

Components are modeled locally in reusable component models including both normal and faulty behavior. Faults are modeled as fault states with unobservable transitions to them, that a component can chose non-deterministically at any time. See Chapter 5.

The diagnosis problem then basically is to infer from the observations the state of the system. Procedures for fault detection and fault isolation using the basic operations of the relational framework are presented in Chapter 7, but the issue of an effective implementation suitable for online diagnosis is only briefly treated.

We then show how to analyze diagnosability properties in Chapter 8. A fault is said to be detectable if there exists a transition in the system model that leads to a detection in a finite number of steps. The transition necessary for detection can be automatically deduced from the system model under certain conditions.

In a system with many possible faults and a limited amount of sensors, it is often the case that the exact fault that has occured cannot be uniquely isolated. Without adding sensors or change something in the system to improve diagnosability, we have to accept that certain faults cannot be distinguished and define a fault partition, i.e., a set of sets of faults. We are then satisfied if we can decide which set in the fault partition the occured fault belongs to. Given a system model, we present a method for deciding the finest possible fault partition under a single fault assumption.

The approach presented is similar to the formal language approach by Sampath et al. reviewed in Section 1.1.2, but there are some fundamental differences, e.g., in how sub-models are synchronized. In exchange of allowing less general models when performing the diagnosability analysis, we get more analysis results than in the formal language approach. The practical value of the approach still needs to be proven by applying it to a real application, but this is a matter of future research.

Chapter 3

The basic relational framework

The presentation in this chapter is closely related to parts of Chapter 2 of Gunnarsson [30]. In this chapter the basic mathematical concept of a relation is introduced.

The definition of the mathematical entity *relation* in this thesis, is directly inspired by the well known binary decision diagram (BDD) representation of large propositional logic formulas. In the framework, we basically manipulate cross-products of finite sets in a manner that we have found very intuitive and that also is very close to the used computer tool.

The original BDD article is [8]. In our implemented tool, a generalization of the binary decision diagram called integer decision diagram (IDD) is used, and the representation is canonical. IDDs were introduced by Gunnarsson [30]. The decision diagram representation is far from being problem free, but is known to work well in many practical cases. The representational and computational issues are further treated in Chapter 4.

3.1 Relations and sets

Only finite domains will be considered in this thesis. Therefore sets and relations are always regarded as finite if nothing else is said.

For the sets S_1 and S_2 we have the standard set operations

$$\begin{aligned} \text{union:} &\quad \mathsf{S}_1 \cup \mathsf{S}_2 \\ \text{intersection:} &\quad \mathsf{S}_1 \cap \mathsf{S}_2 \\ \text{complement:} &\quad \overline{\mathsf{S}_1} \\ \text{set minus:} &\quad \mathsf{S}_1 \setminus \mathsf{S}_2 \\ \text{cross product:} &\quad \mathsf{S}_1 \times \mathsf{S}_2 \end{aligned}$$

Explicit sets are denoted with curly brackets $\{\cdot\}$ and an element of a set will be denoted by $[\cdot]$. As an example of the notation and the cross product, we have, e.g.,

$$\{2,3\} \times \{1,4\} = \{\,[2,1],[2,4],[3,1],[3,4]\,\}.$$

If x is a variable that takes values from the set S, then a specific value of the variable is denoted $x \in \mathsf{S}$. A vector of variables, taking values from the set $\mathsf{S}_1 \times \ldots \times \mathsf{S}_n$, is denoted $[\mathsf{x}_1, \ldots, \mathsf{x}_n]$. A specific value of that vector-variable is denoted $[x_1, \ldots, x_n]$ and is hence an element in $\mathsf{S}_1 \times \ldots \times \mathsf{S}_n$.

Definition 3.1 Universe of Discourse
The *universe of discourse* (UoD), denoted $\mathcal{U}$ or $\mathcal{U}_n$, is the set

$$\mathcal{U}_n = \mathsf{S}_1 \times \cdots \times \mathsf{S}_n$$

where S_i, $1 \le i \le n$, are finite sets. □

The reason for defining the UoD is illustrated by the following example.

Example 3.1 Universe of Discourse
The complement $\overline{\mathsf{S}}$ of the set S can be written as

$$\overline{\mathsf{S}} = \mathcal{U} \setminus \mathsf{S}$$

If $\mathcal{U}$ is not specified we cannot decide the interpretation of $\overline{\mathsf{S}}$. □

Which UoD we use in a given situation, is either explicitly stated or obvious from the context. See also Example 3.3.

Subsets of $\mathcal{U}$ are usually called relations, see for example [28], but we will instead use the term *relation set*, and use the term *relation* for an object representing a relation set, see Definition 3.2 and Definition 3.3.

If we regard a relation set as a piece of information, then the corresponding relation is a structure that we use to *canonically* write this information down in a way suited for computer implementation, see further Chapter 4. A relation is hence a unique representation of a relation set.

Definition 3.2 Relation Set
A *relation set*, denoted $\mathcal{R}$, is a subset of the cross product of n finite sets, i.e.,

$$\mathcal{R} \subseteq \mathcal{U}_n$$

is a relation set over the finite sets $\mathsf{S}_i, 1 \le i \le n$.

Remark: A relation set can be defined over any finite number of sets, i.e., for all $n \ge 1$. □

Definition 3.3 Relation
A *relation* $\mathsf{R}(x_1, \ldots, x_n), \mathsf{R} : \mathcal{U}_n \to \mathbb{B}$ is a representation of the relation set $\mathcal{R} \subseteq \mathcal{U}_n$, where x_i, $1 \leq i \leq n$, are variables that take elements from the sets S_i, $1 \leq i \leq n$, as values.
The relation is defined in terms of the relation set as

$$\mathsf{R}(x_1, \ldots, x_n) = \begin{cases} \texttt{true} & \text{if } [x_1, \ldots, x_n] \in \mathcal{R} \\ \texttt{false} & \text{if } [x_1, \ldots, x_n] \notin \mathcal{R} \end{cases}$$

where $[x_1, \ldots, x_n]$ denotes an element in $\mathcal{R}$, and $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$.

Remark: Since a relation is a canonical representation of a relation set, we have:

$$\mathsf{R}_1 \equiv \mathsf{R}_2 \Leftrightarrow \mathcal{R}_1 \equiv \mathcal{R}_2$$

and specifically

$$\begin{aligned} \mathsf{R}(x_1, \ldots, x_n) &\equiv \texttt{true} \Leftrightarrow \mathcal{R} \equiv \mathcal{U}_n \\ \mathsf{R}(x_1, \ldots, x_n) &\equiv \texttt{false} \Leftrightarrow \mathcal{R} \equiv \emptyset. \end{aligned}$$

□

The relation is Boolean valued (denoted $\mathbb{B}$), i.e., the relation is `true` for elements included in the corresponding relation set and `false` otherwise. From this we have that operators similar to those in propositional logic are well defined on relations. These operators will correspond to the set theoretic operators for the relation sets. See Section 3.2 and especially Definition 3.6.

The reason for separating relations and relation sets are somewhat similar as for separating syntax and semantics. The syntax is the relation and the semantics is the relation set. As we will see in Chapter 4 we can use several different syntaxes for relations but the semantics (the relation sets) remains the same.

Example 3.2 Relation and Relation Set
We have some piece of information which consists of the pairs

$$[\text{red}, \text{apple}], \quad [\text{yellow}, \text{banana}]$$

If we put these pairs in a set we can regard this set as a relation set, which relates two fruits with two colors.

The immediate representation of this relation set is of course to store these pairs in a table or database.

If we let the variables f and c correspond to fruit and color respectively, then the following relation

$$\mathsf{R}(\mathsf{f}, \mathsf{c}) = (\mathsf{f} == \text{apple}) \wedge (\mathsf{c} == \text{red}) \vee (\mathsf{f} == \text{banana}) \wedge (\mathsf{c} == \text{yellow})$$

corresponds to the same relation set as the table. (Details of the logical operators in the formula above will be given in Section 3.2 below.) □

We will continue by defining the projection and embedding operators for relation sets.

Definition 3.4 Projection
Let $\mathcal{R} \subseteq \mathcal{U}_n = S_1 \times \cdots \times S_i \times \cdots \times S_n$ be an arbitrary relation.
The set operator $\pi_{S_i} : \mathcal{U}_n \to S_i \;\; 1 \leq i \leq n$ is defined by

$$\pi_{S_i}(\mathcal{R}) = \{ [s_i] \mid [s_1, \ldots, s_n] \in \mathcal{R} \}$$

and called the *projection of* $\mathcal{R}$ *on* S_i. The set elements are $[s_1, \ldots, s_i, \ldots, s_n] \in \mathcal{U}_n$ and $[s_i] \in S_i$.

Remark: Projection on several finite sets, denoted $\pi_{S_i \times S_j}(\mathcal{R})$ is defined in the obvious way. □

Definition 3.5 Embedding
Let $\mathcal{R} \subseteq \mathcal{U}_n = S_1 \times \cdots \times S_n$ be an arbitrary relation.
The set operator $\epsilon_{S_{n+1}} : \mathcal{U}_n \to \mathcal{U}_n \times S_{n+1}$ is defined by

$$\epsilon_{S_{n+1}}(\mathcal{R}) = \mathcal{R} \times S_{n+1}$$

and called the *embedding of* $\mathcal{R}$ *in* $\mathcal{U}_{n+1}$ $(= \mathcal{U}_n \times S_{n+1})$.

Remark:

- Instead of appending S_{n+1} to $\mathcal{U}_n$ we can put the new set in any position of $\mathcal{U}_n$, even though the index notation will be less clear.
- The definition above can easily be generalized to embedding of relation sets in any number of sets, by using recursion.

□

Note that both the projection and embedding operator return relation sets.

3.2 Relational operations

To manipulate relation sets, the ordinary set operations are used. In the actual implementation, see Chapter 4, relations are manipulated and we have found it most natural to use logic operators for that purpose. We will hence use logical operators in the relation formalism. Logic operations and set operations share the same axiomatic laws (see pages 52 and 109 in [28]) and can therefore be used in a similar way but for different domains.

We will in the following definition use the notation X for the vector-variable $[x_1, \ldots, x_n]$, i.e., by $R(X)$ we mean $R(x_1, \ldots, x_n)$.

Definition 3.6 Relational Operations
For the relation sets $\mathcal{R}_1$, $\mathcal{R}_2$, and the corresponding relations $R_1(X)$ and $R_2(X)$, we have:

Relation	**Relation set**
$R_1(X) \land R_2(X)$	$\mathcal{R}_1 \cap \mathcal{R}_2$
$R_1(X) \lor R_2(X)$	$\mathcal{R}_1 \cup \mathcal{R}_2$
$\neg R_1(X)$	$\mathcal{U} \setminus \mathcal{R}_1 = \overline{\mathcal{R}_1}$
$R_1(X) \land \neg R_2(X)$	$\mathcal{R}_1 \setminus \mathcal{R}_2$
$R_1(X) \rightarrow R_2(X)$	$\overline{\mathcal{R}_1} \cup \mathcal{R}_2$
$R_1(X) \leftrightarrow R_2(X)$	$\mathcal{R}_1 \cap \mathcal{R}_2 \cup \overline{\mathcal{R}_1} \cap \overline{\mathcal{R}_2}$

Remark:

- The statements in the last two rows follow directly from the usual definitions of $\rightarrow$ and $\leftrightarrow$:

$$\begin{aligned} a \rightarrow b &= \neg a \lor b \\ a \leftrightarrow b &= a \land b \lor \neg a \land \neg b. \end{aligned}$$

- A useful fact is that the $\rightarrow$ operator can be used for a set inclusion test:

$$R_1(X) \rightarrow R_2(X) \equiv \texttt{true} \Leftrightarrow \mathcal{R}_1 \subseteq \mathcal{R}_2$$

□

For future use, we define the concept of *covering* and *implicant* for relations.

Definition 3.7 Covering and Implicant
A relation R_1 *covers* a relation R_2, and conversely R_2 is an *implicant* of R_1, iff their corresponding relation sets fulfill that

$$\mathcal{R}_1 \supseteq \mathcal{R}_2.$$

The criterion formulated with relational operations is hence that

$$R_2 \rightarrow R_1 \equiv \texttt{true}$$

is fulfilled.

Remark: The term implicant will, if not otherwise explicitly stated, refer to the case when R_2 represents an *assignment*, to be defined below. □

To check if two relation sets are identical, we can check $R_1 \leftrightarrow R_2 = \texttt{true}$. A more convenient way to check for identical relation sets is though to compare the corresponding relations, since the representation of relations used in the implementation is canonical.

Next an example is given that illustrates how the relational operations are interpreted when used on relations with different Universe of Discourse. The effect is that the corresponding relation sets are embedded in a larger UoD.

Example 3.3 Relational Operations and Embedding
For the three sets S_1, S_2 and S_3 define two arbitrary relation sets $\mathcal{R}_1 \subset S_1 \times S_2$ and $\mathcal{R}_2 \subset S_2 \times S_3$. Let the relations $R_1(x_1, x_2)$ and $R_2(x_2, x_3)$ represent these relation sets. The embedding operator in definition 3.5 is defined for relation sets, and the corresponding operation on relations is simply to interpret the relation as defined over the larger UoD.

Relation	Relation set	UoD
$R_1(x, y)$	$\mathcal{R}_1$	$S_1 \times S_2$
$R_1(x, y)$	$\mathcal{R}_1 \times S_3$	$S_1 \times S_2 \times S_3$
$R_2(x, y)$	$\mathcal{R}_2$	$S_2 \times S_3$
$R_2(x, y)$	$S_1 \times \mathcal{R}_2$	$S_1 \times S_2 \times S_3$
$R_1(x, y) \wedge R_2(y, z)$	$(\mathcal{R}_1 \times S_3) \cap (S_1 \times \mathcal{R}_2)$	$S_1 \times S_2 \times S_3$

□

Substitution of a variable in a relation $R(x_1, \ldots, x_n)$ for a value (or another variable) can be denoted $R(x_1, \ldots, x_i, \ldots, x_n)$ as in (3.1) and (3.2). For notational brevity, we will often write $R[x_i \mapsto x_i]$.

The projection operator π for relation sets has a corresponding relational projection operator, defined in terms of the *quantifier operators*.

Definition 3.8 Relational Quantifier Operations
The quantifier operations $\exists$ and $\forall$ are defined as

$$\exists x_i.\ R(x_1, \ldots, x_n) = \bigvee_{x_i \in S_i} R(x_1, \ldots, x_i, \ldots, x_n) \tag{3.1}$$

$$\forall x_i.\ R(x_1, \ldots, x_n) = \bigwedge_{x_i \in S_i} R(x_1, \ldots, x_i, \ldots, x_n) \tag{3.2}$$

Remark:

- Note that we interpret $\exists$ and $\forall$ as operations mapping one relation to another, i.e., $\exists$ and $\forall$ return relations. This differs from the interpretation in propositional logic [28] but is very common in other research areas such as *Quantifier Elimination Theory*, [2, 19, 52].
- The relation $\exists x_i.\ R(x_1, \ldots, x_n)$ corresponds to the relation set

$$\pi_{S_1 \times \ldots S_{i-1} \times S_{i+1} \ldots \times S_n}(\mathcal{R}).$$

□

The projection operator for relations is then defined in terms of the ∃-operator.

Definition 3.9 Relational Projection Operator
For the relation $\mathsf{R}(x_1, \ldots, x_n)$ representing the relation set $\mathcal{R} \subseteq \mathcal{U}_n = \mathsf{S}_1 \times \cdots \times \mathsf{S}_n$ the relational projection operator corresponding to $\pi_{\mathsf{S}_i}(\mathcal{R})$ in Definition 3.4 is defined as

$$\mathsf{Project}\ x_i.\ \mathsf{R} = \exists x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n.\ \mathsf{R}.$$

Remark: Projection on several variables, denoted $\mathsf{Project}\ x_i, x_j.\ \mathsf{R}$, is defined in the obvious way. □

The quantifier operator ∀ has no corresponding set operator.

Working with relations, it is convenient to have a "solver" that gives the corresponding relation set of a relation. The problem of solving a relation is highly dependent on how we represent relations, which will be further discussed in Chapter 4. For now, only the notation for the solution of a relation is given.

Definition 3.10 Solve
Let the relation set $\mathcal{R} \subseteq \mathsf{S}_1 \times \mathsf{S}_2$ be represented by the relation $\mathsf{R}(x, y)$ where $x \in \mathsf{S}_1$ and $y \in \mathsf{S}_2$. The Solve function

$$\begin{aligned} \mathsf{Solve}(\mathsf{R}(x,y)) &= \{\, [x,y] \in \mathsf{S}_1 \times \mathsf{S}_2 \mid \mathsf{R}(x,y) \,\} \\ \mathsf{Solve}(\mathsf{R}(x,y), x) &= \{\, [x] \in \mathsf{S}_1 \mid \exists y.\ \mathsf{R}(x,y) \,\} \end{aligned}$$

returns the relation sets $\mathcal{R}$ respectively $\pi_{\mathsf{S}_1}(\mathcal{R})$. □

It is often necessary to form the relation representing an equation or set of equations of finite functions. For this purpose we will introduce the infix equal operator $\cdot == \cdot$.

Definition 3.11 Equal Operator
The *equal operator*, used as

$$\mathsf{f}(x) == \mathsf{g}(y)$$

returns the relation representing the equation $\mathsf{f}(x) = \mathsf{g}(y)$, where x and y are vectors of variables and $\mathsf{f}(x)$ and $\mathsf{g}(y)$ are finite functions with the same value domain. The relation $\mathsf{f}(x) == \mathsf{g}(y)$ is `true` for all assignments of values to x and y for which $\mathsf{f}(x) = \mathsf{g}(y)$, `false` otherwise. □

Example 3.4 Equal Operator
Let $\mathsf{S}_x, \mathsf{S}_u$ and S_y be finite sets. Consider the equation set

$$\begin{aligned} x_p &= \mathsf{f}(x,u) \\ y &= \mathsf{g}(x,u) \end{aligned}$$

where the variables belong to the domains $x_p, x \in S_x$, $u \in S_u$, $y \in S_y$ and the functions are $f : S_x \times S_u \to S_x$ and $g : S_x \times S_u \to S_y$. It is represented by the relation

$$R(x, u, y, x_p) = (x_p == f(x, u)) \wedge (y == g(x, u)).$$

The universe of discourse is $\mathcal{U}_4 = S_x \times S_u \times S_y \times S_x$. □

Definition 3.12 Independent variables
Consider a relation $R(x, y_1, y_2)$.
The variables (vectors of variables) y_1 and y_2 are *independent w.r.t.* R if there exist relations $R_1(x, y_1), R_2(x, y_2)$ so that

$$R(x, y_1, y_2) = R_1(x, y_1) \wedge R_2(x, y_2).$$

□

If such relations exist they need not be unique, but one pair can be trivially found:

$$R_1(x, y_1) = \exists y_2.\, R$$
$$R_2(x, y_2) = \exists y_1.\, R.$$

The notation from the definition is used. In general, the relation set of R is a subset of $\exists y_2.\, R \wedge \exists y_1.\, R$. In Example 3.4 the variables x_p and y are independent by construction.

We will now discuss the important embedding aspect (Definition 3.5) of assignment of values to variables. An *assignment* $\mathcal{A}$ of values to variables has two parts; the binding of a unique value to each variable in a set V,

$$\mathcal{A} = \{\, x_i \mapsto x_i \,\}_{x_i \in V}$$

and the complete set of variables for the UoD under consideration. Consider a relation $R(x_1, \ldots, x_n)$ with the UoD $X_1 \times \ldots \times X_n$. We want to assign values, x_i, to the variables in $V \subseteq \{\, x_1, \ldots, x_n \,\}$. The application of the assignment to the relation can be seen as an intersection followed by a quantification of the assigned variables,

$$\exists V.\, R \wedge \bigwedge_{x_i \in V} (x_i == x_i)$$

or as a direct substitution in the relation:

$$R[\,\{x_i \mapsto x_i\}_{x_i \in V}\,].$$

The resulting restricted relation is the same, and the difference is in the implementation; substitution is generally faster, see Chapter 4.

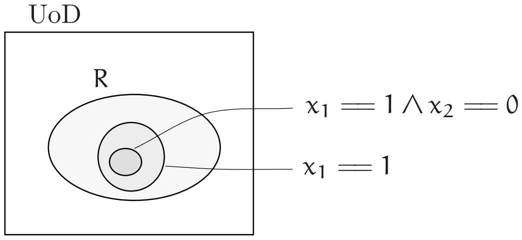


Figure 3.1: Relation set interpretation of covering and implicants, Example 3.5
.

As indicated, the relation representation of an assignment can be written as a conjunction of single variable assignments

$$\bigwedge_{x_i \in V} (x_i = x_i).$$

The variables not in the assignment but in the complete variable set of the UoD, $\{x_1, \ldots, x_n\} \setminus V$, are unspecified and can take any value in their definition domain. In accordance with Definition 3.7, we define the concept of *covering* for assignments. Given two assignments, $\mathcal{A}$ and $\mathcal{B}$ represented by $V_A, V_B \subseteq \{x_1, \ldots, x_n\}$, $\mathcal{A}$ *covers* $\mathcal{B}$, if $V_A \subseteq V_B$ and they assign the same value to all common variables. An assignment, represented by $V \subseteq \{x_1, \ldots, x_n\}$, is said to be *complete* w.r.t. to the UoD, if $V = \{x_1, \ldots, x_n\}$. That is, a value is assigned to all variables.

As already indicated in the remark to Definition 3.7, an assignment $\mathcal{A}$ is an implicant of a relation R, if the relation corresponding to $\mathcal{A}$ is an implicant of R. An assignment $\mathcal{A}$ is a *prime implicant* of R, if $\mathcal{A}$ is an implicant of R, and no other assignment simultaneously covers $\mathcal{A}$ and is an implicant of R.

Example 3.5 Covering and Implicants
The UoD is given by three Boolean valued variables, x_1, x_2, x_3. For example, the assignment $\{x_1 \mapsto 1, x_2 \mapsto 0\}$ is covered by the assignment $\{x_1 \mapsto 1\}$.

Let the relation R represent the table

x_1	x_2	x_3
1	0	0
1	0	1
1	1	0
1	1	1

Then the assignments $\{x_1 \mapsto 1, x_2 \mapsto 0\}$ and $\{x_1 \mapsto 1, x_3 \mapsto 0\}$ are implicants of R and $x_1 = 1$ is a prime implicant of R (the only one, in fact). See also the Venn diagram in Figure 3.1. □

3.3 Interpretations of relations

We will call the interpretation of a variable in a relation as an input respectively output for the *causality* of the variable. The value of an output variable is calculated from the relation given the values of the input variables. Which variables in a relation that are interpreted as outputs and which variables that are interpreted as inputs, is not given by the relation itself, but this causality structure is ultimately decided by the context of the relation, i.e., of how it is used.

If we want to impose a certain causality structure on a relation, it would be good practice to ensure that the relation is *non-partial* w.r.t. to the input (where both the input and output possibly are vectors of variables). A finite function is non-partial w.r.t. a variable, if there exists an output value for each value of the variable. A relation is non-partial w.r.t. a variable is there exists a solution to the relation for each value of the variable. The notion of of non-partialness for finite functions and relations is formally introduced in the following definitions.

Definition 3.13 Non-partial finite function
Let x be a vector of finite variables. A finite function $f(x)$ is said to be *non-partial* if it is defined for all values of the variable(s) x, *partial* otherwise.

Remark: If $D_f(x)$ is the definition domain of $f(x)$ we have

$$D_f(x) = \exists y.\ (y == f(x)).$$

□

We also introduce the corresponding definition for relations.

Definition 3.14 Non-partial relation
Let x and y be vectors of finite variables. The relation $M(x, y)$ is said to be *non-partial w.r.t.* x if

$$\exists y.\ M(x, y) \equiv \texttt{true},$$

partial w.r.t. x otherwise. If $M(x, y)$ is interpreted as a mapping from x to y, then the relation set of $D(x) := \exists y.\ M(x, y)$ is the *definition domain.* □

See Example 3.6.

To demand non-partialness for an input variable in a relation, i.e., to demand that the projection on the input is the whole value range, would eliminate the risk of causality conflicts and also increase reusability, since the relation more safely can be used in different contexts. See further Chapter 5. If the variables that get their definition domain restricted by D are *state* variables, as will be introduced in Chapter 5, it would be sufficient if the reachable states of the system (Definition 6.1) are covered by the restriction.

Definition 3.15 Well formed input causality
Consider a relation $M(u, y)$ with the desired feasible value set $F(x)$. The variable vector u is a *well-formed input* if $D(x)$ from Definition 3.14 fulfills

$$F(x) \to D(x) \equiv \texttt{true}$$

That is, the actual definition domain D for the input u given by the relation M covers the desired feasible input values F. □

A table can be seen as the relational set of a relation, as illustrated in the following example.

Example 3.6 Relations and Tables
Consider the table

x_1	x_2	x_3
0	0	0
1	0	2
1	2	1

where the variables have the value domain $\{0, 1, 2\}$, i.e., $x_1, x_2, x_3 \in S = \{0, 1, 2\}$. The table can be seen as the relational set of the following relation

$$\begin{aligned} R(x_1, x_2, x_3) = &(x_1 == 0) \wedge (x_2 == 0) \wedge (x_3 == 0) \vee \\ &(x_1 == 1) \wedge (x_2 == 0) \wedge (x_3 == 2) \vee \\ &(x_1 == 1) \wedge (x_2 == 2) \wedge (x_3 == 1). \end{aligned}$$

The relation R is non-partial w.r.t. x_3, as can be seen directly in the table or by the fact that $\exists x_1, x_2.\ R = \texttt{true}$. R is though partial w.r.t. x_1 and x_2.

If we apply the assignment $\mathcal{A} = \{x_2 \mapsto 0\}$, then the new relation set is given by the following table:

x_1	x_3
0	0
1	2

□

To conclude Chapter 3, where the notions of relations and relation sets have been introduced, an example is given that illustrates the operators and functions for relations and relation sets.

Example 3.7 Relations and Operators
Let $S_1 = \{0, 1, 2\}$ and $S_2 = \{3, 4\}$ be finite sets with integer elements. The UoD will be the cross product of S_1 and S_2 containing six elements.

$$\begin{aligned} \mathcal{U}_2 &= S_1 \times S_2 \\ &= \{[0, 3], [0, 4], [1, 3], [1, 4], [2, 3], [2, 4]\} \end{aligned}$$

Let $\mathcal{R}_1 = \{[0,4],[1,3]\}$ and $\mathcal{R}_2 = \{[1,3],[2,3]\}$. The corresponding relations will be written as

$$\begin{aligned}
R_1(x_1,x_2) &= (x_1 == 0) \wedge (x_2 == 4) \vee (x_1 == 1) \wedge (x_2 == 3)\\
R_2(x_1,x_2) &= (x_1 == 1) \wedge (x_2 == 3) \vee (x_1 == 2) \wedge (x_2 == 3)
\end{aligned}$$

For the relation operations in definition 3.6 we get

$$R_1(x_1,x_2) \;\wedge\; R_2(x_1,x_2) = (x_1 == 1) \wedge (x_2 == 3)$$

$$\begin{aligned}
R_1(x_1,x_2) \vee R_2(x_1,x_2) = {} & (x_1 == 0) \wedge (x_2 == 4) \vee\\
& (x_1 == 1) \wedge (x_2 == 3) \vee\\
& (x_1 == 2) \wedge (x_2 == 3)
\end{aligned}$$

corresponding to $\mathcal{R}_1 \cap \mathcal{R}_2 = \{[1,3]\}$ and $\mathcal{R}_1 \cup \mathcal{R}_2 = \{[0,4],[1,3],[2,3]\}$.

The complement $\overline{\mathcal{R}_1} = \{[0,3],[1,4],[2,3],[2,4]\}$ corresponds to negating the corresponding relation $R_1(x_1,x_2)$.

$$\begin{aligned}
\neg R_1(x_1,x_2) &= \neg\big((x_1 == 0) \wedge (x_2 == 4) \vee (x_1 == 1) \wedge (x_2 == 3)\big) && (3.3)\\
&= (x_1 == 0) \wedge (x_2 == 3) \vee (x_1 == 1) \wedge (x_2 == 4) \vee\\
&\quad\; (x_1 == 2) \wedge (x_2 == 3) \vee (x_1 == 2) \wedge (x_2 == 4) && (3.4)
\end{aligned}$$

Note that it is not obvious at this point how to derive expression (3.4) from expression (3.3). In fact we have not yet decided how to canonically represent relations. Canonical forms and manipulations of relations will be discussed further in Chapter 4. For now we just observe that (3.3) and (3.4) will have the same value (`true` or `false`) for the same values of x_1 and x_2.

By construction we know that $\mathcal{R}_1 \not\subseteq \mathcal{R}_2$. This can be checked by

$$\begin{aligned}
R_1(x_1,x_2) \rightarrow R_2(x_1,x_2) &= \neg R_1(x_1,x_2) \vee R_2(x_1,x_2)\\
&= /\text{All elements in } \mathcal{U}_2 \text{ except } [0,4]/\\
&= \neg\big((x_1 == 0) \wedge (x_2 == 4)\big)
\end{aligned}$$

The result above is not identically `true` which means that $\mathcal{R}_1$ is not included in $\mathcal{R}_2$.

The projection of $\mathcal{R}_1$ on S_1 is

$$\pi_{S_1}(\mathcal{R}_1) = \{[0],[1]\} \tag{3.5}$$

which corresponds to the existential quantification

$$\begin{aligned}
\exists x_2.\, R_1(x_1,x_2) &= \bigvee_{x_2 \in S_2} R_1(x_1,x_2)\\
&= (x_1 == 0) \vee (x_1 == 1) && (3.6)
\end{aligned}$$

which is a relation only in the variable x_1.

The universal quantifier would in this case be

$$\forall x_2.\ R_1(x_1, x_2) = \texttt{false}$$

since there is no single value of x_1 for all values of x_2 in the relation set $\mathcal{R}_1$.

From the projection (3.5) we make the embedding

$$\epsilon_{S_2}(\mathcal{R}_1) = \{\, [0, 3], [0, 4], [1, 3], [1, 4] \,\}$$

which is also represented by the relation from Equation (3.6)

$$(x_1 == 0) \vee (x_1 == 1) \tag{3.7}$$

but regarded as embedded into the UoD $\mathcal{U}_2$.

Finally, we use the solver function Solve on $R_1(x_1, x_2)$.

$$\mathsf{Solve}(R_1(x_1, x_2), x_2) = \{\, [3], [4] \,\}$$

□

When counting the number of solutions of a relation we usually decide the UoD from the variables that are included in the relation. In the case of relation (3.7) we have a univariate relation with two solutions. If we regard (3.7) as a result from an embedding into S_2 then there are four solutions to the relation.

How to effectively represent relations and compute the operations introduced in this section, is treated in Chapter 4.

Chapter 4

Representing relations

This is a self-contained chapter mainly intended for readers with no prior knowledge of decision diagrams and BDDs.

4.1 Introduction

In this chapter we will consider how relations and the operations on relations introduced in Chapter 3 can be effectively implemented on a computer, with so called *decision diagrams*. The presentation in this chapter is largely a summary of parts of Chapter 3 to Chapter 6 in Gunnarsson [30].

Finite functions and relations are very closely related. The representations that will be presented here are representations of finite functions, and a relation is a function, mapping from some finite domain to the Boolean domain $\mathbb{B}$.

Binary Decision Diagrams (BDD) were introduced by Akers [1] for testing of digital devices. Then Bryant [8] introduced operations and algorithms that utilize the ordering of the variables in BDDs. Bryant showed that in many interesting cases, albeit not all, Boolean functions could be represented by reduced Boolean function graphs in size polynomial to the number of variables[1]. Boolean function graphs are directed acyclic graphs where each vertex has two children (or subgraphs). Bryant also proved that the reduced function graphs were a canonical form for Boolean functions.

After that many different decision diagrams similar to BDDs have emerged, all building on the basic function graph structure. See [10] for an overview of this research area.

Efficient algorithms for Boolean operators were developed by Brace et al. [7]. To extend the Boolean decision diagrams to general finite functions with integer valued value- and definition domains, has been an important task. For example,

[1]The representation could not be polynomial for all Boolean functions, since the well-known satisfiability problem and the problem of deciding whether two logic formulas are equivalent, are solvable in constant time (checking one pointer, respectively comparing two pointers), given the BDD-representation. Both problems are known to be NP-complete.

Srinivasan et al. [75] have developed so called Multi-Valued Decision Diagrams (MDD), serving as a mapping to an underlying BDD representation.

In [30] Gunnarsson introduces a decision diagram, called Integer Decision Diagrams (IDD), not relying on any Boolean structures to represent general finite functions and relations. For the work in this thesis, an IDD implementation developed by Gunnarsson has been used. The implementation is an extension of the BDD implementation from Carnegie Mellon University as described in [7].

In Section 4.2 we will present the fundamental results of reduced function graphs which is the structure from which all decision diagrams are built. The results are due to Bryant [8] and Srinivasan et al. [75]. Section 4.3 introduces the binary decision diagram (BDD) and discusses some of the difficulties in representing general finite domains using only Boolean variables. Some complexity results are also given. The last section 4.4 briefly presents the extension of binary decision diagrams to integer decision diagrams (IDD) made by Gunnarsson [30].

4.2 Function graphs

In this section we give the basic results that all decision diagrams are based on. A function graph is a data structure that can be used to efficiently and canonically represent finite functions in a computer. It is basically a rooted, directed acyclic graph.

Definition 4.1 Function Graph

i) A *function graph* is a rooted, directed graph with vertex set V containing two types of vertices, *nonterminal* and *terminal*.

   A *nonterminal* vertex ν has as attributes an index denoted $\mathsf{index}(\nu) \in \{1, \dots, n\}$ where $n \in \mathbb{Z}_+$, a value range denoted $\mathsf{range}(\nu) \in \mathbb{Z}_+$ and children vertices denoted $\mathsf{child}(\nu, i) \in \mathsf{V}$, $0 \leq i < \mathsf{range}(\nu)$.

   A *terminal* vertex ν has as attribute a value denoted $\mathsf{value}(\nu) \in 0, \dots, \mathsf{M} - 1$, where $\mathsf{M} \in \mathbb{Z}_+$.

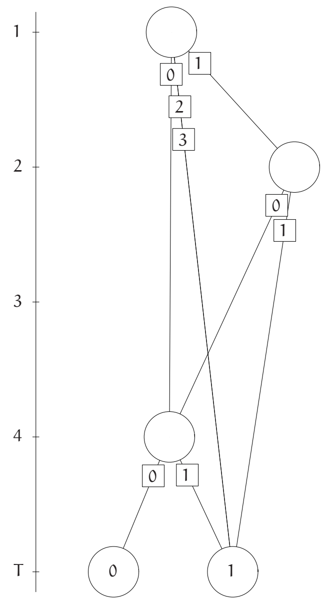
ii) For any nonterminal vertex ν, if $\mathsf{child}(\nu, i)$ is also nonterminal, then $\mathsf{index}(\nu) < \mathsf{index}(\mathsf{child}(\nu, i))$.

iii) The value range $\mathsf{range}(\nu)$ is equal for all vertices having the same index.

Remark: Note that the restriction on indices makes the function graphs acyclic since any path from the root vertex to one of the terminal vertices must have strictly increasing index values. □

To illustrate the definition, we give a simple example of a function graph from Gunnarsson [30].

Example 4.1 Function Graph
We give a simple function graph.



The vertical axis to the left shows the indices for the vertices. In Definition 4.3, each index on the vertical axis corresponds to a variable in the finite function the graph represents. We denote a vertex by $v_{i,j}$ where i is the index and j is the order in the picture starting from left with $j = 1$. That is, the vertices in the above graph are $v_{1,1}, v_{2,1}, v_{4,1}, v_{T,1}$ and $v_{T,2}$.

The numbers in the square edge labels in the figure identifies the children of a vertex. If there are more than one label on the same line, then there are several edges connecting the same pair of vertices. The terminals are placed at the bottom of the graph and indicated by a T on the vertical axis. The values of the terminals are written inside the vertices. T is not an index but is used as one for notational simplicity.

To further connect to Definition 4.1, we have the following for the function graph above:

$$
\begin{aligned}
\mathsf{index}(v_{1,1}) &= 1 \quad \text{the root}\\
\mathsf{child}(v_{1,1},0) &= v_{4,1}\\
\mathsf{child}(v_{1,1},1) &= v_{2,1}\\
\mathsf{child}(v_{1,1},2) &= v_{T,2}\\
\mathsf{child}(v_{1,1},3) &= v_{T,2}\\
\mathsf{child}(v_{2,1},0) &= v_{4,1}\\
\mathsf{child}(v_{2,1},1) &= v_{T,2}\\
\mathsf{child}(v_{4,1},0) &= v_{T,1}\\
\mathsf{child}(v_{4,1},1) &= v_{T,2}\\
\mathsf{value}(v_{T,1}) &= 0\\
\mathsf{value}(v_{T,2}) &= 1\\
\mathsf{range}(v_{1,1}) &= 4\\
\mathsf{range}(v_{2,1}) &= 2\\
\mathsf{range}(v_{4,1}) &= 2
\end{aligned}
$$

Note that there are no vertices with index 3 for this function graph, which means the function represented by the function graph do not depend on the third variable. □

Since a function graph is an acyclic directed graph, every vertex roots a subgraph which in turn is a function graph of its own.

Definition 4.2 Subgraph
For any vertex v in a function graph G, the *subgraph rooted by* v is defined as the graph consisting of v and all of its descendants. □

Having defined the structure of function graphs we will establish the correspondence between function graphs and functions.

Definition 4.3 Function Graph to Function Connection

A function graph G having a root vertex v, $\mathsf{index}(v) = 1$ and maximum index of vertices n defines a function $f_v(x_1, \dots, x_n)$. With $u = v$ initially the recursive definition is

1. If u is a terminal vertex, then $f_u = \mathsf{value}(u)$.
2. If u is a nonterminal vertex with $\mathsf{index}(u) = i$, then f_u is the function

$$f_u(x_i, x_{i+1}, \dots, x_n) = f_{\mathsf{child}(u,x_i)}(x_j, \dots, x_n) \tag{4.1}$$

where $j = \mathsf{index}(\mathsf{child}(u, x_i))$ and the variable $x_i \in \{0, 1, \dots, \mathsf{range}(u) - 1\}$.

□

To be able to form function graphs with canonical properties we need some definitions before we can state the theorem of canonical function graphs.

Definition 4.4 Isomorphic Function Graphs
Let G and G' be function graphs with vertex sets V and V' respectively. G and G' are *isomorphic* if there exists a bijective function $\sigma : V \to V'$ such that for any vertex $v \in V$ with $\sigma(v) = v'$, either both v and v' are terminal vertices with

$$\mathsf{value}(v) = \mathsf{value}(v')$$

or both v and v' are nonterminal vertices with

$$\begin{aligned}\mathsf{index}(v) &= \mathsf{index}(v'),\\ \mathsf{range}(v) &= \mathsf{range}(v')\end{aligned}$$

and

$$\sigma(\mathsf{child}(v, i)) = \mathsf{child}(v', i) \quad 0 \leq i < \mathsf{range}(v).$$

□

The mapping σ is in fact quite constrained since the root vertex of G must map to the root of G', and the order of the children must be preserved all the way down to the terminals.

The following lemma is rather immediate.

Lemma 4.1 Isomorphic Subgraphs
If G is isomorphic to G' by the mapping σ, then for any vertex v in G, the subgraph rooted by v is isomorphic to the subgraph rooted by $\sigma(v)$. □

The following definition gives us the *reduced function graph*, which is the theoretical base for the canonical representation of relations used in this thesis.

Definition 4.5 Reduced Function Graph
A function graph G is *reduced* if it contains no vertex v with

$$\mathsf{child}(v, 0) = \mathsf{child}(v, 1) = \cdots = \mathsf{child}(v, \mathsf{range}(v) - 1),$$

nor does it contain distinct vertices v and v' such that the subgraphs rooted by v and v' are isomorphic. □

Lemma 4.2 Reduced Subgraphs
For every vertex v in a reduced function graph, the subgraph rooted by v is itself a reduced function graph. □

The following theorem is the main result of this section and shows that a reduced function graph is a canonical form for the corresponding function.

Theorem 4.3 Reduced Function Graph is a Canonical Form
For any function f over a finite domain, there exists a unique (up to isomorphism) reduced function graph representing f, and any other function graph representing f contains more vertices. □

Proof The theorem is proved by Bryant [8] for the Boolean case only, but the outline of that proof can easily be used in this case as well, which has been done by Srinivasan et al. [75]. ■

4.3 Binary Decision Diagram (BDD)

For the case of Boolean functions and variables we use a special form of the reduced function graphs called *binary decision diagram* (BDD) [8]. In Definition 4.1 the range for all nonterminal vertices v are restricted to $\mathsf{range}(v) = 2$ and there are only two terminal vertices, i.e., $M = 2$ in Definition 4.1.

If we have a Boolean expression $f(x_1, \ldots, x_n)$, the connection between function graph and function in Definition 4.3 corresponds in the Boolean algebra to the famous *Shannon expansion*

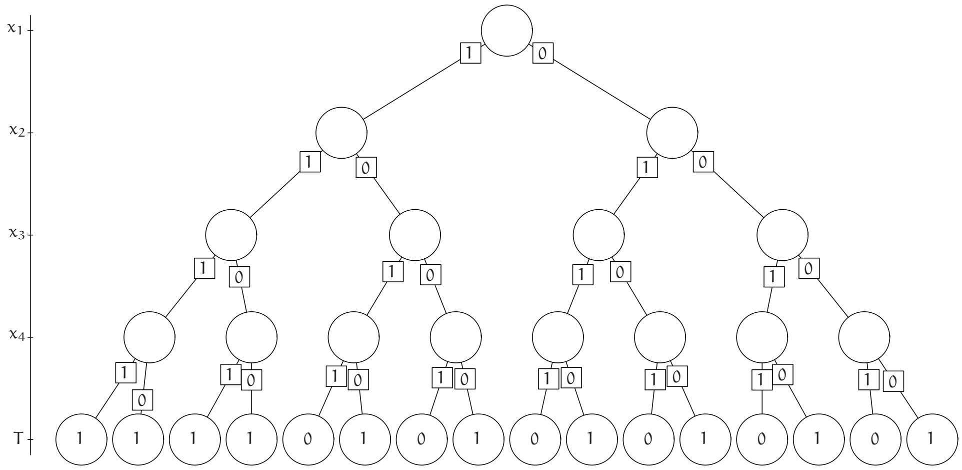
$$f(x_1, \ldots, x_n) = x_1 f(1, x_2, \ldots, x_n) + \neg x_1 f(0, x_2, \ldots, x_n). \tag{4.2}$$

If we continue this expansion as far as it goes, we get a new Boolean expression representing the same Boolean function as f. The BDD then basically represents this expression with common subexpressions eliminated.

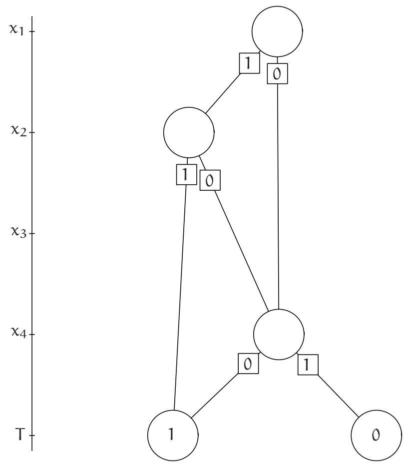
If we change the order in which we expand w.r.t. the variables we usually get very large differences in the number of vertices of the reduced BDD.

Germundsson [27] shows that a BDD can be viewed as a representation of a polynomial in a polynomial ring $\mathbb{R}_2[x_1, \ldots, x_n]$ using a certain basis function, the so called Lagrange interpolation function, and a reduction of common subexpressions.

Example 4.2 [30] Unreduced vs. Reduced BDD
The Boolean function $f = x_1 \wedge x_2 \vee \neg x_4$ is represented by the following ordered but unreduced binary tree if we use the variable ordering $x_1 < x_2 < x_3 < x_4$.



After reduction we get the following BDD



Note that since the function f does not depend on the variable x_3, it disappears in the reduced BDD. □

Algorithms to compute the Boolean operations ($\wedge, \vee, \neg$, etc.), quantification, restriction (i.e., insertion of values), composition and solution are given in [7] and can also be found in [30]. The algorithms explore the graph structure, maintains the canonical form and are all polynomial in the size of the involved BDDs.

For example, the `And` ($\wedge$) and `Or` ($\vee$) operations between two BDDs F and G have the worst case complexity $\mathcal{O}(|\mathsf{F}||\mathsf{G}|)$, where $|\cdot|$ denotes the number of vertices. The `Not` ($\neg$) operation is done in constant time, whereas existential and universal quantification, if we quantify on *all* variables in the BDD F, has worst case complexity $\mathcal{O}(|\mathsf{F}|^3)$.

Since all algorithms are polynomial in the number of vertices, the critical parameter for the BDDs to be usable in practice, is the size of the BDD graphs. Here we can only point to the many successful applications, where BDDs have been found to be a tractable structure for representing and doing computations on Boolean functions [49, 9, 31]. The choice of a good variable ordering is critical and different heuristics and "rules of thumb" exists for making this choice, see the cited literature. The basic rule is that inputs should be ranked lower than outputs. The intuition is that the outputs depend on the inputs, and should be found higher up in the tree structure.

The BDD implementation we are using is developed at the Carnegie Mellon University and is described in [7]. Here at the Automatic Control department at Linköping University, an interface to the general purpose computing environment *Mathematica*[2] has been developed using the MathLink protocol.

4.3.1 Using Binary Decision Diagrams

In Chapter 3 no restrictions on the domain of variables were made, except that they were finite. When using BDDs we only have Boolean variables, so a variable $z \in \mathbb{Z}_r$ where $r > 2$ and $\mathbb{Z}_r = \{ 0, \dots , r-1 \}$, must be represented by several Boolean (BDD) variables $b_1, \dots , b_q$ as a binary number. The number of BDD variables needed for the variable z is q, where q is the smallest positive integer fulfilling

$$2^q \geq r. \tag{4.3}$$

We then regard the BDD variables $b_1, \dots , b_q$ as one integer valued variable z. Since we only can represent domains with cardinality a power of two, equality might not be attained in (4.3). This means that the domain the BDD variables can represent as bits in a binary number is larger then the domain of the variable z that they are representing. This might cause problems if not certain care is taken. A simple example will illustrate the problem.

Example 4.3 Representing relations with BDDs
Consider the variable $z \in \mathbb{Z}_3$ and the simple relation

$$R(z) = (z == 0) \vee (z == 2). \tag{4.4}$$

The solution set of R is $\mathsf{Solve}(R, x) = \{ [0], [2] \}$.

This relation is encoded using the BDD variables $b_1, b_2 \in \mathbb{B}$ as

$$R_{\text{BDD}} = (\neg b_1 \wedge \neg b_2) \vee (\neg b_1 \wedge b_2) \tag{4.5}$$

if we let b_2 represent the more significant bit.

The complement $\neg R(z)$ should have the solution set $\{ [1] \}$, but if we do the negation and solving computations on R_{BDD} using the BDD package and then translate

[2]Trademark of Wolfram Research [81]

the binary result back to integers, the solution set is $\{ [1], \underline{[3]} \}$, which is clearly wrong since $3 \notin \mathbb{Z}_3$.

Another problem that might occur is if we take the disjunction between $\mathsf{R}(z)$ and another relation with no dependence on z. For example, $\mathsf{R}(z) \vee \mathsf{P}(y)$, where $\mathsf{P}(y) = y$ and $y \in \mathbb{B}$. The expected solution set is

$$\{ [z, y] : [0, 0], [0, 1], [2, 0], [2, 1], [1, 1] \}, \tag{4.6}$$

but what we actually get when performing these computations with the BDD package is the solution set

$$\{ [z, y] : [0, 0], [0, 1], [2, 0], [2, 1], [1, 1], \underline{[3, 1]} \} \tag{4.7}$$

□

The safe solution is to perform all operations on the relations represented by BDDs modulo the desired UoD (Definition 3.1). We introduce the *lambda polynomial* for this purpose.

Definition 4.6 Lambda polynomial
Given the UoD $\mathbb{Z}_{r_1} \times \ldots \times \mathbb{Z}_{r_n}$ and the variables $z_i \in \mathbb{Z}_{r_i}$ we define a *lambda polynomial* for each variable as

$$\lambda_{q_i}^{r_i}(z_i) = \bigvee_{j=0}^{r_i - 1} (z_i == j), \; i = 1, \ldots, n. \tag{4.8}$$

where q_i are the smallest positive integers fulfilling

$$2^{q_i} \geq r_i, \; i = 1, \ldots, n. \tag{4.9}$$

The *lambda polynomial* for the whole UoD is then defined as

$$\Lambda(z_1, \ldots, z_n) = \bigwedge_{i=1}^{n} \lambda_{q_i}^{r_i}(z_i). \tag{4.10}$$

Remark:

- We will use the terms lambda polynomial and *lambda relation* synonymously. The term lambda polynomial were introduced by Gunnarsson and Plantin [29, 60] for similar reasons as here when representing finite functions as polynomials over finite fields.

- All the lambda polynomials, or lambda relations, would just be the trivial relation `true` if we could represent the relations in a structure directly suited to the UoD. The IDD introduced in Section 4.4 is such a structure.

□

The lambda polynomial, or lambda relation, $\Lambda(z_1, \dots, z_n)$ is the restriction of the domain the BDD structure is capable of representing, to the desired UoD.

With the modification that we have to restrict the relations represented and computed with BDDs to the desired UoD with the lambda polynomial Λ, all operations introduced in Chapter 3 can be performed with the BDD package.

To conclude this section, we continue Example 4.3.

Example 4.4 Representing relations with BDDs, cont.
All relations in this example are assumed to be represented as BDDs using BDD variables, although we will use the variable z for notational convenience.

The lambda polynomial for the UoD $\mathbb{Z}_3 \times \mathbb{B}$ is

$$\begin{aligned}\Lambda(z) &= (z == 0) \vee (z == 1) \vee (z == 2) \\ &= (\neg b_1 \wedge \neg b_2) \vee (b_1 \wedge \neg b_2) \vee (\neg b_1 \wedge b_2)\end{aligned} \tag{4.11}$$

The variable y is not present in Λ since $y \in \mathbb{B}$ and hence can be exactly represented with a BDD variable.

The restriction of R with Λ makes no difference,

$$R(z) = R(z) \wedge \Lambda(z) \tag{4.12}$$

but by restricting the relation resulting from the negation of R, we get the expected solution set $\mathsf{Solve}(\neg R \wedge \Lambda, z) = \{ [1] \}$.

By restricting $P(y) = y$ to the correct UoD

$$P_{\text{res}}(y, z) = y \wedge \Lambda(z) \tag{4.13}$$

the disjunction with $R(z)$ also gives the expected result

$$\mathsf{Solve}(R \vee P_{\text{res}}, \{ z, y \}) = \{ [0, 0], [0, 1], [2, 0], [2, 1], [1, 1] \}. \tag{4.14}$$

□

If all relations are restricted with the lambda polynomial to the correct UoD, then the only operation that can render illegal solutions is the negation of a relation, i.e., the complement of the corresponding relational set. The lambda polynomial needs to be applied after the negation.

4.4 Integer Decision Diagram (IDD)

The *integer decision diagram* (IDD) introduced by Gunnarsson [30] is an extension of the binary decision diagram that implements the general function graphs in Section 4.2. Hence an IDD can represent any function $f : \mathbb{Z}_{r_1} \times \dots \times \mathbb{Z}_{r_n} \to \mathbb{Z}_r$ where $r, r_1, \dots, r_n \in \mathbb{Z}_+$. With $r = 2$, f is a relation as introduced in Section 3.

An IDD representing a Boolean valued function, i.e., a relation, will be called a *Boolean IDD* ($\mathbb{Z}_2$ and $\mathbb{B}$ are the same from the IDD point of view).

Since the value range can be chosen freely for each variable we do not need to bother with bit-representation of integer valued variables and the use of lambda polynomials described in Section 4.3.1 is unnecessary. Moreover, since fewer variables are needed to represent a relation, we hope that the IDDs will prove to have a complexity advantage over BDDs, both in graph size and algorithmically. This will be a matter of further research, but see [30] for a benchmark.

Gunnarsson [30] has made an efficient IDD implementation in a C code package, which is linked to *Mathematica* and integrated with our tools for modeling and analysis of DEDS. The algorithms for effective computation of the Boolean operators, quantification, restriction, composition and solution mentioned in the previous section 4.3 about BDDs, are extended to Boolean IDDs with the same worst case complexity.

A number of new operations concerning integer arithmetic and relational operations (i.e., $<$ and $>$) for general IDDs are also introduced and implemented exploring the graph structure.

Chapter 5

Modeling DEDS with Relations

In this chapter, the basic concepts of DEDS system theory in the relational framework is introduced. The chapter closely follows and partly extends Chapter 2 in [30].

5.1 Introduction

The IFAC Technical Committee SAFEPROCESS has proposed a broad definition of a *model* of a system:

> "a static or dynamic relation among system variables and parameters that describes the systems behavior in quantitative mathematical terms."

To be more specific on the type of system we are interested in, Ramadge-Wonham [63] has formulated a definition of discrete event systems in one sentence:

> "A Discrete Event System (DES) is a dynamic system that evolves in accordance with the abrupt occurrence, at possibly unknown irregular intervals, of physical events."

To stress the dynamics of these systems even further they are often referred to as Discrete Event Dynamic Systems (DEDS). From the definition above we have that a DEDS takes events as inputs. The dynamic of the DEDS depends on events to appear if any transition of the system should take place.

We will use a slightly different approach. We let the DEDS react on signals, i.e., the inputs and outputs of a DEDS are values changing as time elapses. The DEDS is *measuring* the values of the input at the time instants when the system makes a transition. This is very similar to reactive transition systems, see further Section 5.5.

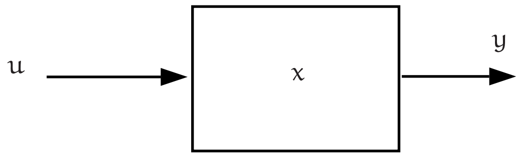


Figure 5.1: Input, output and state variables of a DEDS.

This approach is quite natural if we, e.g., regard DEDS in which a computer is polling some sensors or operator signals and reacting on these measures in some way. The setups of control systems are often made in this way, even though there are cases where the event driven approach is more appropriate.

Modeling frameworks of a DEDS can vary in syntax and features. Common frameworks for modeling DEDS are, e.g., *finite automata* and *Petri Nets*, that have properties that make them easy to use and to interpret due to their convenient graphical appearances. See [30] for a comparative review.

Our aim is not not to try to introduce a new modeling syntax or language to model DEDS, but focus on the mathematics of DEDS and how to represent DEDS in a more mathematical way, see also [30, 27]. What we want is a conceptually simple representation of DEDS that is suitable for computer representation and computation and that captures the relevant properties. Our stab at this is to use the relation from Definition 3.3 and a signal based *relational model* introduced below.

5.2 Relational models

The relational framework introduced in Chapter 3 and Chapter 4, is the basic representational and computational tool we use to model and work with discrete event dynamic systems.

Definition 5.1 DEDS variables
Consider Figure 5.1. The *input* $\mathsf{u} \in \mathsf{U}$ is a received signal to the DEDS where U is the *input domain*. The *state* $\mathsf{x} \in \mathsf{X}$, where X is the *state domain* or *state space*, contains the information that must be remembered between each transition to describe the system behavior.

The *output* $\mathsf{y} \in \mathsf{Y}$ is a produced signal from the DEDS, and Y is the *output domain*. □

We will refer to whether a variable is regarded as an input or output as the *causality* of the variable.

The perhaps most important aspect of a dynamic system is the state, and how it evolves over time. A DEDS takes a *transition* when a new value for the state x is chosen. The DEDS we work with are untimed and we are only interested in the possible sequences of transitions. The interval between two transitions is called a *time step*. In the sequel, we will assume that each state variable models a *component*, or subsystem, of the DEDS under consideration.

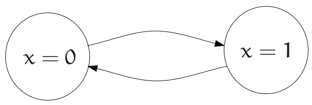
The dynamics of a DEDS is represented by a *transition function* or a *transition relation*, that maps the input u and current state x to the next state, denoted by x^+. In the same manner, the I/0-behavior of a DEDS is given by an *I/O-function* or *I/O-relation*, that maps the inputs and current state to the output y.

Definition 5.2 Transition and I/O mapping
The *transition function* $f : X \times U \to X$, $x^+ = f(x, u)$, and the *transition relation* $M_t : X \times U \times X \to \mathbb{B}$, $M_t(x, u, x^+)$, both represent the dynamic behavior of the DEDS, i.e., how the next state depends on the input and current state.

The *I/O function* $g : X \times U \to Y$, $y = g(x, u)$, and the *I/O relation* $M_{io} : X \times U \times X \to \mathbb{B}$, M_{io}(x,u,y), both represent the I/O behavior of the DEDS, i.e., how the output depends on the input and current state. □

Consider the following example:

Example 5.1 Flip Flop



The state transition function and the transition relation for the toggling DEDS in the above figure is

$$x^+ = f(x, u) = \neg x$$

respectively

$$M_t(x, x^+) = (x^+ == \neg x).$$

□

We now introduce the entity we will use to represent and work with DEDS in this thesis.

Definition 5.3 Relational model
The *relational model*, $M(z, z^+)$, is a relation representing a mapping from the the *system variables* z to the next system variables z^+.

Remark: In general, the system variables z and z^+ denote a vector of variables as

$$z = [z_1, \dots, z_n]$$
$$z^+ = [z_1^+, \dots, z_n^+],$$

although they are often partitioned in state, input and output variables, i.e.,

$$
\begin{aligned}
z &= [x, u, y] \\
z^+ &= [x^+, u^+, y^+].
\end{aligned}
$$

□

We will sometimes use the terms variable and signal interchangeably, but variable refers more to the entity in the relation, and signal refers more to the value taken by the variable.

The relational model $M(z, z^+)$ does not necessarily depend on all variables in z and z^+. Specifically, the relational model may be *asymmetric*, meaning that the only variables in z^+ that are present in the relation M are the state variables. Conversely, the relational model is called *symmetric*, if it has the same definition domain for z^+ as for z.

Definition 5.4 Symmetric/asymmetric relational model
Consider a relational model $M(z, z^+)$ where the system variables are partitioned in state, input and output variables, i.e., $z = [x, u, y]$ and $z^+ = [x^+, u^+, y^+]$. The relational model M is *asymmetric* iff

$$M(z, z^+) = M(x, u, y, x^+).$$

We define the relations

$$
\begin{aligned}
R(z) &= \exists z^+.\ M(z, z^+) \\
R^+(z^+) &= \exists z.\ M(z, z^+).
\end{aligned}
$$

M is *symmetric* iff

$$R(z) = R^+(z)$$

□

A relational model can without loss of information be converted between asymmetric and symmetric ($z = [x, v]$):

$$
\begin{aligned}
M_{\text{asym}}(x, v, x^+) &= \exists v^+.\ M_{\text{sym}}(x, v, x^+, v^+) \\
M_{\text{sym}}(x, v, x^+, v^+) &= M_{\text{asym}}(x, v, x^+) \wedge R(z^+)
\end{aligned}
$$

Some differences when computing reachable states are discussed in Section 6.2. The asymmetric form reduces the number of variables in the relation, which is an advantage for representational reasons, see Chapter 4. When not otherwise explicitly stated, a relational model is always asymmetric.

In Figure 5.2, we show a conceptual view of a DEDS described by an asymmetric relational model. The box with q^{-1} denotes a one step delay, and hence the actual memory.

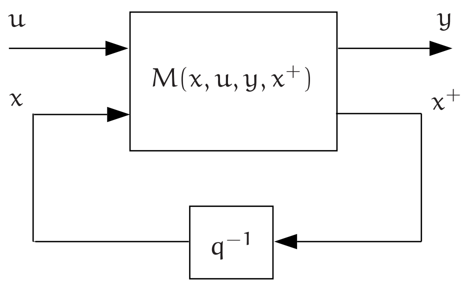


Figure 5.2: A DEDS described with a relational model M

To connect to the Definition 5.2 of transition and I/O mappings, we define the *transition behavior relation* and *I/O behavior relation* for an asymmetric relational model M:

$$\begin{array}{rcl} M_t(x, u_t, x^+) & = & \exists y.\, M \\ M_{io}(x, u_{io}, y) & = & \exists x^+.\, M. \end{array} \tag{5.1}$$

The inverse relationship is $M = M_t \wedge M_{io}$, that holds if y and x^+ are independent (Definition 3.12). Note that not all inputs necessarily affect both M_t and M_{io}. The inputs affecting the transition behavior are called *transition inputs*, and are denoted u_t, the inputs affecting the I/O behavior are called *ordinary inputs* (for lack of better terminology), and denoted u_{io}. The two input vectors u_t and u_{io} need not be distinct.

5.3 Determinism/Nondeterminism

A deterministic DEDS has only one possible behavior given the current state and input whereas a nondeterministic DEDS may choose to take any of several possible trajectories. Compare with deterministic/nondeterministic finite automata (DFA/NFA) [37].

A DEDS model, Definition 5.3, is said to be *deterministic* if the value of the next state variable x^+ is uniquely specified for all (feasible) values of the state variable x and input u, *nondeterministic* otherwise. In Section 5.9 we will use nondeterminism to model unobservable events, which in our case will be faults.

When a DEDS model is used to obtain values of non-measurable variables in a real system, i.e., as an observer, nondeterminism can give rise to uncertainty on which values state and other variables have in the real DEDS. Other sources of uncertainty are non-measurable inputs and incompletely specified initial state.

We say that a DEDS model is *deterministically observable* if the system state can be known at all times given an initial state, known inputs, measured outputs and transition times. Note that a nondeterministic system might be deterministically observable, if the suitable component states or outputs are measurable, and that a deterministic DEDS might not be deterministically observable if, e.g., all inputs are not known. Aspects of observability will be further discussed in Chapter 7, in the context of fault detection and isolation. DEDS observability is treated in detail by, e.g., Özweren, Willsky [82] and Ramadge [62].

5.4 DEDS component model

The view in Definition 5.3 of a DEDS is useful when you have a complete model of a system, and use it for different kinds of analysis and synthesis, see, e.g., [31, 30, 27].

We will in Section 5.8 propose a modeling scheme for a certain physical domain, with local component models that can be connected to form a complete system model. To that end, we describe a DEDS as a number of concurrently operating components that interact via shared variables.

In the following definition, we propose a structure of a DEDS component model, refining and specializing the structure in the remark to Definition 5.3. The main purpose of this structure, together with the rules for connection of models in the following sections, is to be an aid when designing reusable local component models for physical systems.

Definition 5.5 DEDS component model
A DEDS component model consists of several variables together with a transition relation and two relations for component connections. The variables are partitioned into the following disjunct sets: ($x, x^+, u_t, u_I, u_a, y_I, y_a$ and a can all be vectors of variables):

- *A state (or mode) variable.* Usually only one per component, and here denoted x, respectively x^+ for the next state.
- *Transition inputs*, affecting which transitions that are possible. Denoted u_t.
- *Information variables.* Variables intended for information passing between components, e.g., control and sensor signals. Here denoted u_I respectively y_I.
- *Connection (or shared) variables.* Variables intended to model physical connections of components. Usually denoted a, u_a and y_a. The variables a are *acausal*, i.e., no fixed input/output interpretation is specified. They are usually used for physical entities.

The relations are:

- *The transition relation* $M_t(x, u_t, x^+)$, interpreted as a mapping from x, u_t to x^+. M_t is non-partial w.r.t. $[x, u_t]$, i.e., $\mathsf{Project}\ x, u_t.\ M_t \equiv \texttt{true}$.
- *The information relation* $M_I(x, u_I, y_I)$, interpreted as a mapping from x, u_I to y_I. M_I is non-partial w.r.t. $[x, u_I]$, i.e., $\mathsf{Project}\ x, u_I.\ M_I \equiv \texttt{true}$.
- *The connection (shared variables) relation* $M_a(x, u_a, a, y_a)$.

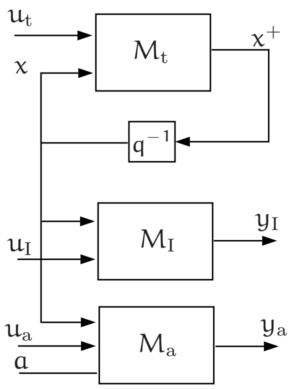


Figure 5.3: The general structure of a DEDS component model.

Remark:

- The shared variables relation M_a has deliberately no non-partialness requirement.
- The (asymmetric) relational model in Definition 5.3 is obtained as

$$M(z, z^+) = M_t(x, u_t, x^+) \wedge M_I(x, u_I, y_I) \wedge M_a(x, u_a, a, y_a),$$

where $z = [x, u_t, u_I, y_I, u_a, a, y_a]$.

□

Note the Moore machine structure (see, e.g., [37]) of the DEDS component model, where the input/output is bound to the states and not the transitions. The causality structure of the definition is graphically illustrated in Figure 5.3.

The rationale of the classification of variables and the partitioning of the relational model from Definition 5.3 is mainly to make clear how component models are to be composed to form the complete system model, see Section 5.5 and Section 5.6. When connecting component models agreeing with Definition 5.5, the goal is often to obtain a relational model as in Definition 5.3, with the variables partitioned in states, inputs and outputs. In Section 5.8 we present a detailed example, where it also is exemplified why we cannot have a non-partialness requirement on all relations in the component model.

A DEDS model on the form of Definition 5.5, is deterministic if the next state x^+ is uniquely specified for all feasible values of the state x and transition input u_t.

5.5 Composition of models

When combining several DEDS models to a single model we say that we take the *composition* or form a *product* of the models. In this thesis we will only use *synchronous composition*. Other possible compositions are *asynchronous* and *interleaved*, that allow a looser ordering of the transitions between submodels. All these compositions are treated in the relational framework in [30].

Definition 5.6 Synchronous Product
The *synchronous product* ($\|$) of the relational models $M_1(z_1, z_1^+)$ and $M_2(z_2, z_2^+)$ is

$$M_1(z_1, z_1^+) \parallel M_2(z_2, z_2^+) = M_1(z_1, z_1^+) \wedge M_2(z_2, z_2^+) \tag{5.2}$$

□

This definition requires all subsystems (components) to take a transition simultaneously. This differs somewhat from the definition in e.g., [37] and the Ramadge-Wonham framework [63] where only transitions with common events are synchronized. It is consistent, though, with the component composition for transition systems described in [79]. As pointed out there, this synchronization is correct if the real order in which the components make their transitions , do not influence the final state. This will henceforth be assumed.

If a component should not change state in a system transition, there must hence exist a transition in the component model back to the same state. We can with careful modeling obtain a similar synchronization as in the Ramadge/Wonham framework, and model the same kind of underlying real behavior.

The state of the composed system, i.e., the current value of all state variables, will be called the *system state*, or system configuration. We will call the individual state variables *component state variables*, and their individual values in a certain system state *component states*.

5.6 Connecting models

The models we put together with the synchronous product, communicate via shared variables. As discussed in Section 3.3, the variables in a relational model do not have an inherent causality, and what is calculated from what is decided by how the relation is used. The design decision to allot a certain causality for a variable in a component model is mostly information on the intended use of the model. If certain care is not taken with the causality of shared variables between the submodels, conflicts may arise where two submodels restrict the same variable to different values, which means that the system state is no longer a solution of the systems relational model. A sufficient requirement is that the relational model is non-partial w.r.t. its inputs.

An input variable of a component is usually connected to an output variable of another component. The non-partialness ensures that the input variable can take

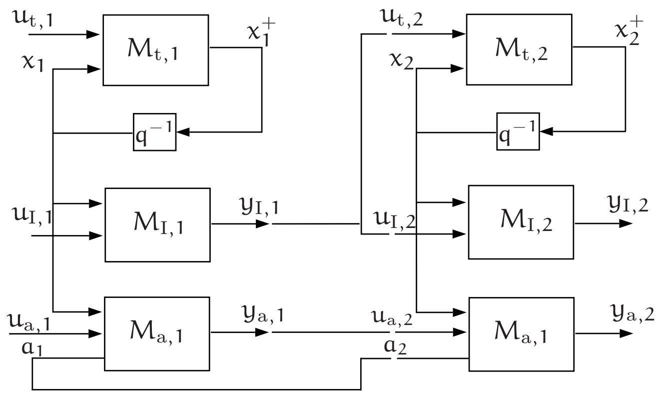


Figure 5.4: Two connected DEDS component models.

any value from its definition domain without placing restrictions on the output variable it is connected with. The output variable on the other hand can bind the input variable it is connected to, to certain values. For example, in Definition 5.5, it is required that the transition relation M_t and the information relation M_I are non-partial w.r.t. their respective inputs.

How two component models can be connected is illustrated in Figure 5.4. We let $y_{I,1}$ consist of the two single variables $y_{I,1}^1$ and $y_{I,1}^2$, the first one connecting to $u_{t,2}$, the second one to $u_{I,2}$. For simplicity, we let all other variables be single variables. The relational model of the connected system could then be written

$$M(z, z^+) = (M_1 \| M_2) \wedge (u_{t,2} \equiv y_{I,1}^1) \wedge (u_{I,2} \equiv y_{I,1}^2) \wedge \\ (u_{a,2} \equiv y_{a,1}) \wedge (a_1 \equiv a_2)$$

where $z = [x, u_t, u_I, a, y_I]$ if we define the variables $x = [x_1, x_2], u_t = [u_{t,1}, u_{I,1}]$, $u_I = u_{I,1}, u_a = u_{a,1}, y_a = y_{a,2}, a = a_1$ and $y_I = [y_{I,1}^1, y_{I,1}^2, u_{t,2}, u_{I,2}, u_{a,2}, a_2, y_{I,2}]$.

There are some remarks to make. First, in this connection, the input $u_{I,1}$ affects via $M_{I,1}$ both the ordinary output and the transitions of the new model, which is indicated by that it is classified both as u_t and u_I. It is still a perfectly valid DEDS model according to Definition 5.3, but we have made a minor violation of the structure in Definition 5.5. Whether this should be allowed, or if the Moore machine structure should be kept by allowing $M_{I,1}$ to connect to either $M_{t,2}$ or $M_{I,2}$ ($M_{a,2}$) but not to both, is a design decision.

Second, if the variable a_1 (a_2) should be classified as an output after the connection and not an acausal variable, is a question that must be answered on a case to case basis.

Third, the variables used in the connection are now internal to the model and are seen as outputs. We can, without loss of information, remove duplicates using existential quantification:

$$M_r(z_r, z_r^+) = \exists u_{t,2}, u_{I,2}, u_{a,2}, a_2.\ M$$

The only difference between z and z_r, is that y_I is reduced to $y_{I,r} = [y_{I,1}^1, y_{I,1}^2, y_{I,2}]$.

To get the model on the same form as in Definition 5.5, we can define

$$\begin{aligned}
M_t(x, u_t, x^+) &= \exists y_{I,1}^1, u_{t,2}.\ M_{t,1} \wedge M_{t,2} \wedge (u_{t,2} == y_{I,1}^1) \wedge M_{I,1} \\
M_I(x, u_I, y_I) &= \exists y_{I,1}^2, u_{I,2}.\ M_{I,1} \wedge M_{I,2} \wedge (u_{I,2} == y_{I,1}^2) \\
M_a(x, u_a, y_a, a) &= \exists a_2, u_{a,2}, y_{a,1}.\ M_{a,1} \wedge M_{a,2} \wedge \\
&\quad\ (a_1 == a_2) \wedge (u_{a,2} == y_{a,1})
\end{aligned} \tag{5.3}$$

where $x = [x_1, x_2], u_t = [u_{t,1}, u_{I,1}], u_I = u_{I,1}, u_a = u_{a,1}, y_a = y_{a,2}, a = a_1$ and $y_I = y_{I,2}$. The relation $M_{I,1}$ is needed to get M_t because of the connection between $y_{I,1}^1$ and $u_{t,2}$. We have here also chosen to completely eliminate the internal variables, instead of having them as outputs.

We can also interpret the connected components as a DEDS model according to Definition 5.3 and equation (5.1) with $M_t(x, u_t, x^+)$ as in (5.3) and

$$M_{io}(x, u_{io}, y) = M_I \wedge M_a,$$

by letting $x = [x_1, x_2], u_t = [u_{t,1}, u_{I,1}], u_{io} = [u_{I,1}, u_{a,1}]$ and $y = [y_{a,2}, a_1, y_{I,2}]$.

In Section 5.8, we show with a detailed example how a class of physical systems can be modeled with reusable component models in a local fashion with carefully designed mappings for the shared acausal variables.

To conclude this section, we go through the details of a simple example.

Example 5.2 Interconnection of models

Consider the two DEDS whose transition relations are visualized in Figure 5.5. The input/output relations are defined:

$$\begin{aligned}
M_{I,1}(x_1, y_{I,1}) = {} & (x_1 == 0) \to (y_{I,1} == 0) \wedge \\
& (x_1 == 1) \to (y_{I,1} == 1)
\end{aligned}$$

and

$$\begin{aligned}
M_{I,2}(x_2, u_{I,2}, y_{I,2}) = {} & (x_2 == 0) \to (y_{I,2} == 0) \wedge \\
& (x_2 == 1) \to (y_{I,2} == u_{I,2}) \wedge \\
& (x_2 == 2) \to (y_{I,2} == 0).
\end{aligned}$$

There are no physical connections or acausal variables in this example, and hence no relation M_a. The transitions not annotated with $u_{t,2}$ in DEDS 2 in Figure 5.5 can be taken regardless of the transition input. Also note that DEDS 2 is nondeterministic.

We now want to connect these two models by making the left one control the transitions of the right one, as shown in Figure 5.6. We form the connected model

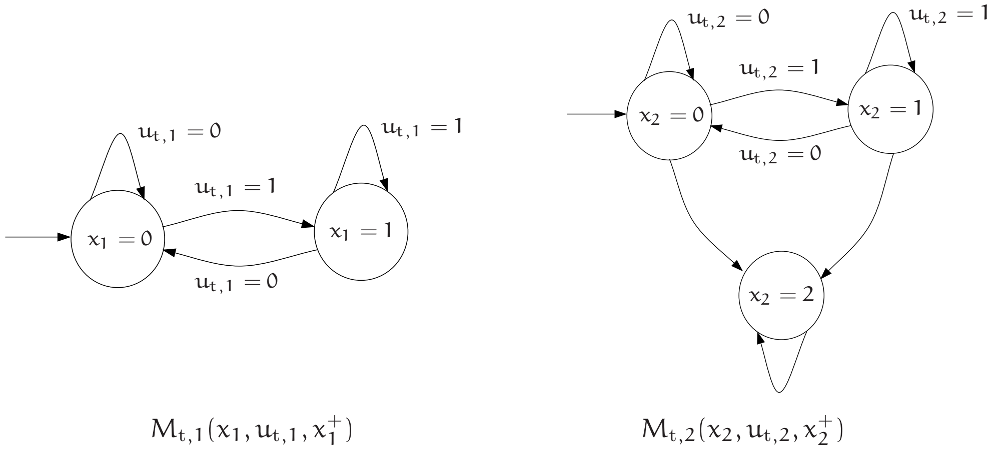


Figure 5.5: The transition relations of two DEDS.

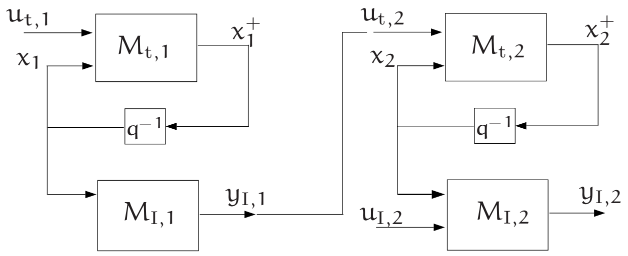


Figure 5.6: The two DEDS component models connected in Example 5.2.

$$M_t(x, u_t, x^+) = \exists u_{t,2}, y_{I,1}.\ (M_{t,1} \| M_{t,2}) \wedge M_{I,1} \wedge (u_{t,2} = y_{I,1}) \tag{5.4}$$
$$M_I(x, u_I, y_I) = M_{I,2} \tag{5.5}$$

or simply

$$M(x, u_t, u_I, y_I, x^+) = \exists u_{t,2}, y_{I,1}.\ (M_1 \| M_2) \wedge (u_{t,2} = y_{I,1}) \tag{5.6}$$

where $x = [x_1, x_2], u_t = u_{t,1}, u_I = u_{I,2}$ and $y_I = y_{I,2}$. The internal variables $y_{I,1}$ and $u_{t,2}$ have been quantified for simplicity. The alternative is to interpret them as outputs from the new model. The transition relation for the resulting DEDS is visualized in Figure 5.7. The two system states where DEDS 2 is in state $x_2 = 2$ have, to keep the figure simple, been collapsed to one state, marked [*,2]. $\square$

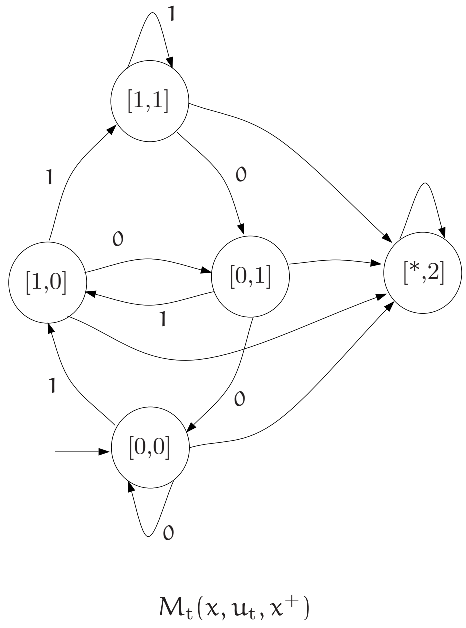


Figure 5.7: Transition relation for the connected DEDS in Example 5.2.

5.7 Restricting models

It is often useful to constrain the behavior of a system $M(z, z^+)$ or $M(x, u, y, x^+)$ in certain ways. We may want to remove certain states and/or certain transitions. To restrict a model so that certain forbidden states cannot be reached is common in synthesis methods for DEDS, see e.g., Ramadge-Wonham [63] for the formal language approach and [27, 30] for a treatment in the relational framework. We will use restriction of behavior when analyzing detectability and isolatability in Chapter 8.

We make the following definition.

Definition 5.7 Restriction of Models
Let $M(z, z^+)$ be a relational model and $R(z)$ a relation representing some property. We define the operator $\mathsf{Restrict}[M(z, z^+), R(z)](z, z^+)$ that returns a relational model with symmetrically restricted behavior as follows

$$\mathsf{Restrict}[M, R](z, z^+) = R(z) \ \wedge \ M(z, z^+) \ \wedge \ R(z^+)$$

If the relational model is asymmetric, i.e., $M(x, u, y, x^+)$, and the restricting relation is $R(x, u, y)$, the asymmetric definition is

$$\mathsf{Restrict}[M, R](x, u, y, x^+) = R(x, u, y) \ \wedge \ M(x, u, y, x^+) \ \wedge \ R^+(x^+)$$

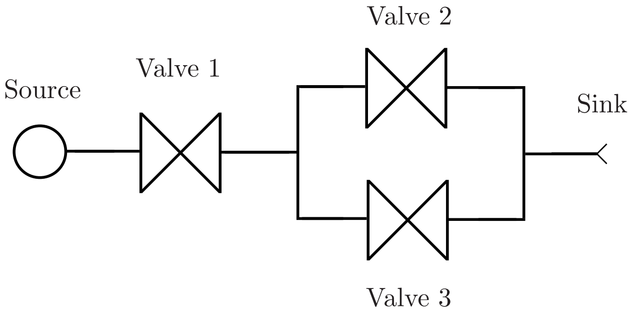


Figure 5.8: A flow/pressure system.

where

$$\mathsf{R}^{+}(x) = \mathsf{Project}\, x.\ \mathsf{R}(x, u, y).$$

□

The restriction of the next state variables for the asymmetric definition is necessary, since otherwise transitions leading to states outside $\mathsf{R}(z)$ could still be present in the resulting model.

5.8 Example of local physical modeling

We propose a modeling method for systems consisting of connected pipes, valves and sensors, with flow and pressure as interesting physical entities. Such systems will be referred to as flow/pressure systems. See Figure 5.8 for an example. The components used in Figure 5.8 are valves, sources, sinks and junctions. The modeling method consists of relational models and connection rules for these components.

Each component will be modeled with a number of variables and mappings between them, according to the structure in Definition 5.5. The physical variables are *flow* and *pressure*, usually denoted with f and p, that models what a flow/pressure sensor in the system would register. In terms of Definition 5.5, they are acausal variables. When a complete system is composed, the physical variables at the topological positions of sensors are interpreted as outputs.

It is assumed that there is only one-way flow through the real pipes and valves, from left to right in Figure 5.8. The allowed topological direction of flow will be called downstream, and the other direction upstream. We of course also need to assume that flow and pressure are meaningfully modeled as Boolean variables, that is, it is sufficient to model presence/absence of flow and pressure.

The condition for flow to be present at a certain point in a flow/pressure system, is that there is an open path downstream from a source to a sink. To capture this condition in local component models, we propose the use of *flow-possible* variables, denoted `fp`, that models whether a certain topological point in the flow/pressure system has an open path downstream to a sink. In terms of Definition 5.5, these variables are connection variables, but we specify a fixed causality. The downstream flow possible variables for a component are inputs (u_a) and the upstream variables are outputs (y_a).

The desired properties for pressure modeling is not as clear as for flow, but depends on the application and choice of discretization of pressure measurements. If there is no flow anywhere, then all points in the flow/pressure system with an open path to a source should sense pressure. Another natural property, that might not be suitable for all applications, concerns when no pressure is sensed; If there is no flow anywhere, pressure sensors at positions where flow is possible, i.e., with an open path to a sink, should sense no pressure.

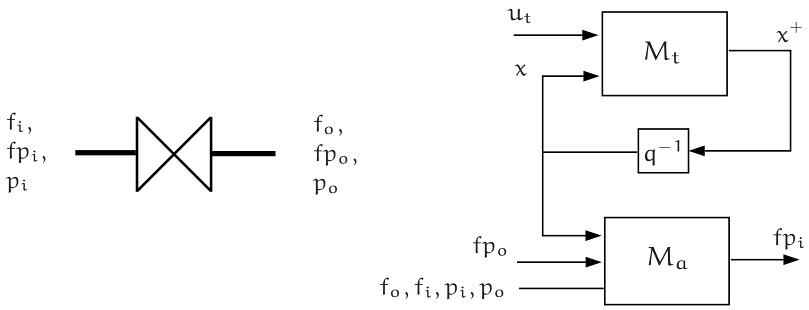
The two properties are in conflict if there is flow present. A position with a path to a sink should have no pressure vs. a position with a path to a source should have pressure. For example, what should happen to a pressure sensor at a position with no flow, but connected to positions with flow?

In an application where, e.g., the flow resistance in the pipes or sinks are high, the former property should be the predominant, i.e., a position with an open path to a source should have pressure. This is what is achieved with the proposed component models. For other systems, it might depend on how close to the outlet the sensor is, and of course on the discretization of the pressure. At positions without any open paths to a source or sink, the pressure is left unspecified. Modeling of pressure, and other entities, can of course be done on a system per system basis using, e.g., global sensor maps, see Section 5.10.

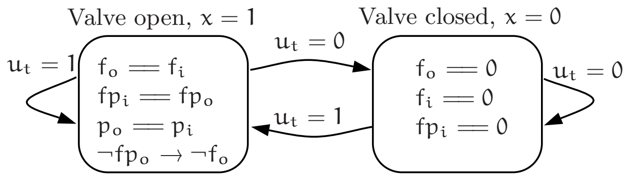
The components and the variables and relational models used to model them are shown in Figure 5.9 to 5.13. Note that the flow possible variables are interpreted as inputs for all component models, but they are not non-partial in the relational model, because of the connection to the flow variables. It is necessary to design the components carefully, so that they cooperate correctly. For example, a flow variable cannot be `true` when the corresponding flow possible variable is `false`.

Two components are connected by composing their relational models and setting their topologically corresponding flow, flow possible and pressure variables equal. The topologically corresponding variables are indicated in the (a) subfigures in Figure 5.9 to 5.13, where the downstream direction is from left to right. Note the use of the flow possible variables to carry information about possible paths from a sink to a source, and the immediate appearance/disapperance of flow along the whole path.

As discussed before, many variables can be quantified from the relational model without losing any information, but reducing the size of the representation of the relation (see Chapter 4). Variables immediately removable are the duplicates of the connection variables flow, flow possible and pressure in each connection. Often we are only interested in having the measurable entities in the model, which means that all flow possible variables, and all flow/pressure variables at positions in the system with no corresponding sensors, can be removed. It is recommendable to remove

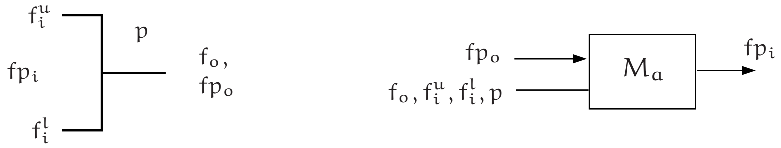


(a) Physical component (b) Causality



(c) Relations M_t and M_a

Figure 5.9: Valve component model

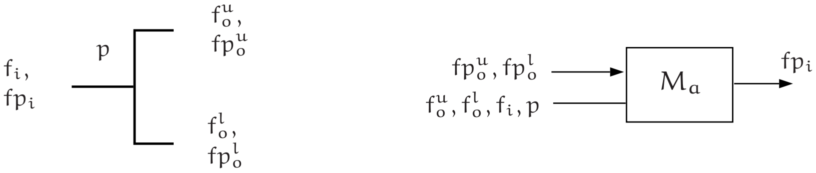


(a) Physical component (b) Causality

$$\begin{aligned}
&f_o == f_i^u \vee f_i^l \\
&fp_i == fp_o \\
&\neg fp_o \to \neg f_o
\end{aligned}$$

(c) Connection relation M_a

Figure 5.10: Junction together component model



(a) Physical component (b) Causality

$$
\begin{aligned}
f_o^u &== f_i \wedge fp_o^u \\
f_o^l &== f_i \wedge fp_o^l \\
fp_i &== fp_o^u \vee fp_o^l \\
\neg fp_i &\rightarrow \neg f_i
\end{aligned}
$$

(c) Connection relation M_a

Figure 5.11: Junction apart component model

(a) Physical component (b) Causality

$$
\begin{aligned}
f &== fp \\
p &== 1
\end{aligned}
$$

(c) Connection relation M_a

Figure 5.12: Source component model

(a) Physical component (b) Causality

$$
\begin{aligned}
fp &== 1 \\
\neg f &\rightarrow \neg p
\end{aligned}
$$

(c) Connection relation M_a

Figure 5.13: Sink component model

intermediate variables as early as possible when putting the system together, to keep down the size of the IDD-representation (see Chapter 4).

To put the model of, e.g., the system in Figure 5.8 together, is simple but tedious. Assume that we have flow and pressure sensors between Valve 1 and the first junction. The model of the two junctions, Valve 2, 3 and the sink has already been composed and intermediate variables have been quantified,

$$\begin{aligned}&M_{t,2}(x_{v,2}, x_{v,3}, u_{t,2}, u_{t,3}, x^+_{v,2}, x^+_{v,3})\\&M_{a,2}(x_{v,2}, x_{v,3}, f_i, fp_i, p_i),\end{aligned}$$

where f_i, fp_i, p_i are the in-variables for the left junction. Valve 1 and the source has the relational models

$$\begin{aligned}&M_{t,1}(x_{v,1}, u_{t,1}, x^+_{v,1})\\&M_{a,1}(x_{v,1}, f_{i,1}, fp_{i,1}, p_{i,1}, f_{o,1}, fp_{o,1}, p_{o,1}),\end{aligned}$$

respectively

$$M_{a,s}(f_s, fp_s, p_s)$$

where the exact relations are given in Figure 5.9 and Figure 5.12 respectively. If we for notational convenience define the connection relation

$$C_a = ([f_{i,1}, fp_{i,1}, p_{i,1}] == [f_s, fp_s, p_s]) \wedge ([f_{o,1}, fp_{o,1}, p_{o,1}] == [f_i, fp_i, p_i])$$

and the intermediate variables $v = [f_{i,1}, fp_{i,1}, p_{i,1}, f_s, fp_s, p_s, f_{o,1}, fp_{o,1}, p_{o,1}, fp_i]$, then the final model, now in terms of Definition 5.3, can be put together with

$$\begin{aligned}M_t(x, u_t, x^+) &= M_{t,1} \| M_{t,2}\\M_{io}(x, y) &= \exists v.\ (M_{a,1} \| M_{a,2} | M_{a,s}) \wedge C_a\end{aligned}$$

where x is the three valve state variables, u_t the three valve transition inputs and $y = [f_i, p_i]$ is flow and pressure between Valve 1 and the first junction.

We will now investigate some restrictions in the modeling power of the local component models presented above. Observe that the model is not valid if a source has a connection with a sink that is partly upstream, as is illustrated with a simple example in Figure 5.14. There is a conflict between the source setting pressure to 1 and the sink, since there is no flow there, setting the pressure to 0 (Figure 5.12(c) and 5.13(c)). This conflict causes the relational model of the system not to have a solution for system states (configurations) where the source and the upstream sink has an open path. In other words, M_a for the whole system is partial w.r.t. the state variables.

We can remove the conflict and the non-partialness by, e.g., leaving the pressure at a sink unspecified, but that would not make the models behavior agree with the real behavior. The modeling of flow proposed here does not capture the possibility of upstream flow. The difficulty in modeling two-way flow locally, manifests itself as the need to reverse causality for the flow possible variables to find *any* open

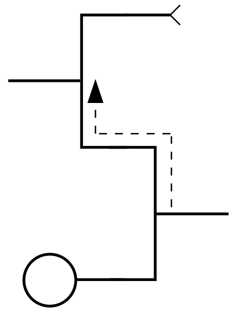


Figure 5.14: Flow/pressure system with conflict caused by upstream flow path.

path from a source to a sink, and not only the downstream paths. It is not known to the author if it is even possible to model this more general situation in a local manner. For this more general modeling problem, one can use global modeling, and, e.g., use graph theoretic methods to find all possible paths from sources to sinks for a certain topology. Since our representation of relations is canonical, the end result will be the same as if local modeling was employed.

5.9 Fault modeling

We are interested in the discrete, untimed behavior of complex systems consisting of interacting components, where faults that affect the system behavior can occur in the components. The faults are abrupt, but do not necessarily bring the system to a stop.

Faults will be modeled using the structure and notation in Definition 5.5, where component faults are modeled by adding *fault states*, or fault modes, to the definition domain of the state variable (x) for the component. The component can chose nondeterministically to enter a fault state at any time and the fault states are mutually exclusive, since the component can be in only one state at a time. The faults will be modeled as persistent, i.e., a fault mode cannot be left once it is entered.

The non-faulty behavior and states of the component, will be called *normal*. In each fault state, the behavior of the component is modeled with M_I and M_a in the same manner as for the normal states. How this models can be used for diagnosis will be discussed in Chapter 7.

As an example, it is straightforward to extend the valve component model presented in Section 5.8 to incorporate the faults stuck open and stuck closed, see Figure 5.15. Note that the faults are modeled as persistent. This extended valve model will be used for examples in subsequent chapters.

As a further example of fault modeling, we propose a component model of a pipe with two possible faults, leak and break in Figure 5.16. This model is not very

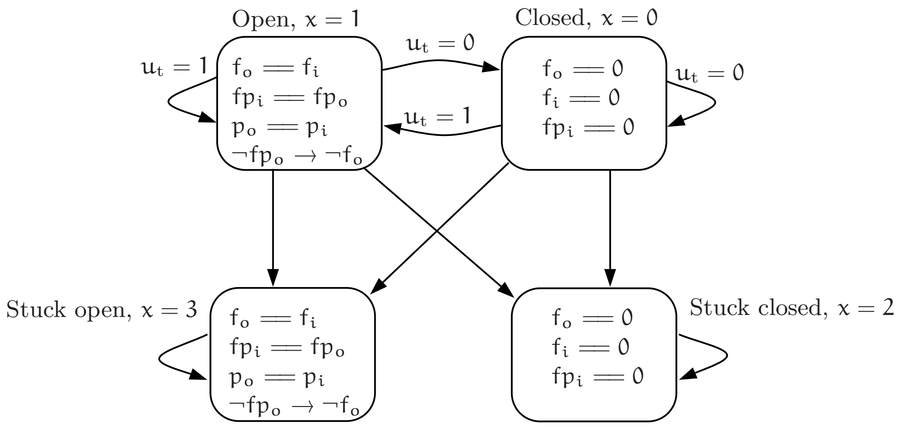


Figure 5.15: Valve component model with modeled faults. Relations M_t and M_a.

powerful, though, since the limitations of this approach mentioned above, severely limits our ability to express what happens physically when a leak or break appears.

A leak is assumed to make flow possible, or more specifically, measurable, even if there is no open path to a sink, but still allow flow to pass the pipe if there is an open path to a sink. Hence it works as a junction apart component with an adapted sink on one branch and the ordinary connection on the other. The adaption of the sink is that the pressure is left unspecified to avoid the conflict between source and sink mentioned above.

A break is assumed to make flow possible even if there is no open path to a sink, and no flow can pass the break. The pressure is once again left unspecified to avoid pressure conflicts that would invalidate the model.

A consequence of the limitations of the flow/pressure modeling, is that the behavior of the real system when there is a leak or break resulting in upstream flow is not described by the model. The diagnosis obtained with the method presented in Chapter 7 will not contain the correct fault, or simply be empty.

To at least get an indication of where the unknown fault is located in the system, the standard solution is to introduce an unknown fault state for the components, where the I/O behavior is completely unspecified. The modeling aspects of this is though not further treated in this thesis. Introducing unknown fault states of course results in more possible diagnoses to discriminate among, see further Chapter 7 for this aspect.

5.10 Global sensor maps

Sensors in a system can measure many entities – flow, pressure, voltage, temperature etc. – and it might be difficult to find local component models that completely

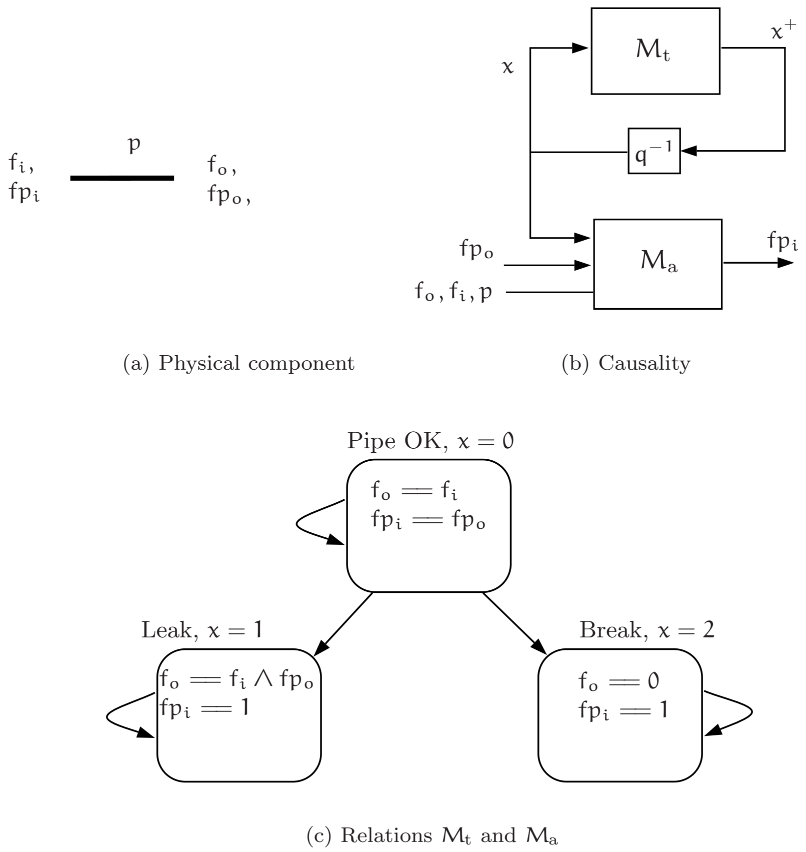


Figure 5.16: Component model of pipe with two possible faults.

capture the mapping from system states and inputs to sensor readings, i.e., outputs.

For example, the component models introduced in Section 5.8 locally model the mapping from system states to flow- and pressure sensors, but only for systems where no sink has an upstream connection with a source.

A global sensor map for a sensor in a system is a mapping, in our framework represented as a relation, from the system states to the sensor readings. We will demonstrate the concept with an example. The simple system modeled could just as well have been modeled with the method in Section 5.8, by adding a pump component model as an active source. The example will be used in the sequel to demonstrate concepts in diagnosis and diagnosis analysis, so a controller scheme is also added. The system is adapted from [70].

Example 5.3 Valve/Pump system
The system is shown in figure 5.17 and consists of a valve, a pump, a sensor that

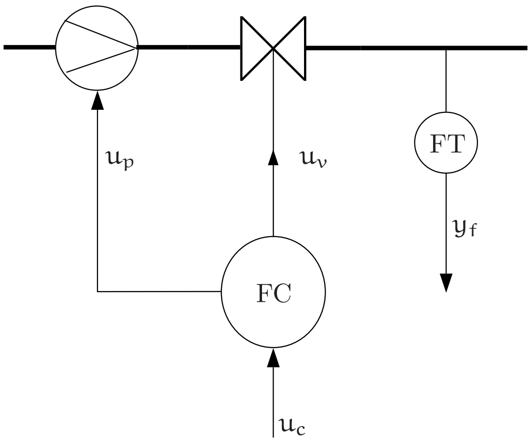


Figure 5.17: Flowchart of a simple system.

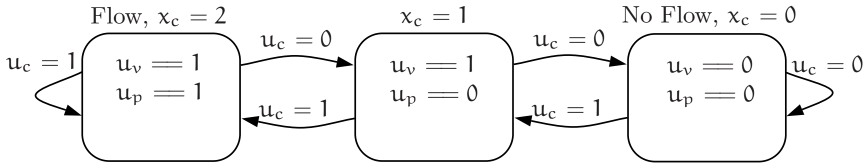


Figure 5.18: Controller for the valve/pump system in Example 5.3. Relation $M_c = M_{t,c} \wedge M_{io,c}$

indicates presence or absence of flow, and a controller. In the figure, the controller is denoted FC (Flow Control) and the sensor is denoted FT (Flow Transmission). The pump and the valve can be either on/off respectively open/closed. The valve and the pump can also fail in two different ways, stuck open or stuck closed, respectively stuck on, stuck off .

The component models of the pump and valve are identical to the pure transition part M_t of the valve model in Figure 5.15. The variables used for the component models are $z_v = [x_v, u_v]$ and $z_p = [x_p, u_p]$, a state variable and a transition input respectively. We hence have the relational models $M_v = M_{t,v}(x_v, u_v, x_v^+)$ and $M_p = M_{t,p}(x_p, u_p, x_p^+)$. Instead of numbers for the values of the state variables, the more suggestive notation $x_v \in \{\mathrm{O}, \mathrm{C}, \mathrm{SO}, \mathrm{SC}\}$ and $x_p \in \{\mathrm{On}, \mathrm{Off}, \mathrm{SOn}, \mathrm{SOff}\}$ will be used.

The specification for the controller scheme is to give respectively cut-off flow according to an external (boolean) input, and not turn the pump on when the valve is closed, since that might damage the pump. A DEDS model of such a controller is shown in Figure 5.18.

The transition inputs of the valve and pump component model is already set equal to the suitable output from the controller, so a first system model is, for example, composed as

$$\tilde{M}(x, u, x^+) = \exists u_v, u_c.\, M_v || M_p || M_c,$$

where $x = [x_c, x_v, x_p]$ and $u = u_c$.

There is only one physically sensible global sensor map, and it can be written down as a table or with your favorite logic formulas; the relation is always the same. Let $y = y_f$ be the boolean output from the flow sensor, and S(x,y) the sensor map:

$$\begin{aligned} F(x) &= (x_v == \mathrm{O} \lor x_v == \mathrm{SO}) \land (x_p == \mathrm{On} \lor x_p == \mathrm{SOn}) \\ S(x, y) &= (F \to y_f) \land (\neg F \to \neg y_f). \end{aligned}$$

The complete system model is now

$$M(x, u, y, x^+) = \tilde{M} \land S.$$

□

Chapter 6

Analysis

Having a model of the DEDS behavior we can perform different types of analysis. A natural distinction to make is between static and dynamic analysis. By static analysis we mean analysis performed on the model in one single iteration, and by dynamic analysis we mean analysis performed on a systems dynamical behavior in any number of steps. The formulation of the analysis in the relational framework done in this chapter is, again, closely related to Gunnarsson [30].

In the following, we assume that $\mathsf{M}(z, z^+)$ is a given relational model of a DEDS.

6.1 Static Analysis

In a relational model, we have the whole system behavior represented in one single object. All possible transitions and all static relations between system variables are contained in the single relational model $\mathsf{M}(z, z^+)$. Hence we can investigate properties of all possible state/input combinations, i.e., all transitions of a DEDS in one go. We call this *static analysis*. The analysis performed in Chapter 8 is to a large part static. Below some brief examples of static analysis are presented. A short glance at Definition 3.6 could be helpful.

If the relation $\mathsf{Q}(z, z^+)$ represents a property or specification, then M fulfills Q iff

$$\mathsf{M}(z, z^+) \to \mathsf{Q}(z, z^+) = \texttt{true}. \tag{6.1}$$

If (6.1) is not fulfilled, then the relation

$$\neg(\mathsf{M}(z, z^+) \to \mathsf{Q}(z, z^+))$$

gives the behavior for which Q is not fulfilled.

Static analysis can be used to test if certain relations between system variables are included in the model or not. Assume that we have a relational model with states, inputs and outputs $\mathsf{M}(x, u, y, x^+)$ and are interested in the states and inputs that give a certain output represented by $R_y(y)$. This can be obtained by

$$\exists y, x^+ .\ M(x, u, y, x^+) \wedge R_y(y)$$

We can also check for deadlock states. Under the assumption that the system $M(x, u, x^+)$ has a well defined transition for all inputs and states, the deadlock states, i.e., the states that the system can never leave, are given by

$$\forall u.\ M(x, u, x)$$

If the relation is `false`, then the system has no deadlock states.

Suppose x are the state variables in the relational model $M(z, z^+)$. If $Q(x)$ represents a subset of the state space, then the model $M(z, z^+)$ is defined for all states in this set if

$$Q(x) \rightarrow \mathsf{Project}\ x.\ M(z, z^+) = \mathtt{true}$$

where we use existential quantification to eliminate all variables except the state variables before the implication is tested.

6.2 Dynamic Analysis

By dynamic analysis we mean analysis that takes any number of iterations of the model into account. Dynamic (or multiple step) analysis gives more information about the behavior of the system than static analysis. In this thesis, the computation of all reachable states from an initial state or set of states is the basic dynamic analysis method. Other, more sophisticated dynamic analysis like temporal logic verification will not be considered. See instead e.g., [49, 31, 30, 27].

The computation of reachable states for discrete event systems is of course a well known analysis method, and is here formulated in the relational framework. Given a model $M(z, z^+)$ we can compute the set of states $R_k(z)$ reachable in k steps or less from some initial set of states $I(z)$

$$\begin{aligned} R_0(z) &= I(z) \\ R_k(z) &= R_{k-1}(z) \vee (\exists \tilde{z}.\ (R_{k-1}(\tilde{z}) \wedge M(\tilde{z}, z))). \end{aligned} \tag{6.2}$$

In each iteration we compute a relation representing all states that are reachable in one step from the states reached so far represented by $R_{k-1}(z)$. This set is added by union (relation operation $\vee$) to $R_{k-1}(z)$. If the relational model M is asymmetric, as explained in Section 5.7, only the pure state variables will be present in R_i. If the relational model is symmetric, also any mapping between the input-output variables will be present in the reachable states.

Since we are dealing with finite state systems this iteration will eventually reach a fixed point, i.e., $R_d(z) = R_{d+1}(z)$ for some finite d. The minimal number of steps d needed to reach the fixed point is the *depth* of the system. In most engineering applications the depth is far below its maximal possible value, which is 2^n for a system described by n binary state variables. An example of a system with maximal depth is a counter. The maximal possible number of reachable states are likewise 2^n.

In [31] an industrial sized case study is described, where the relational framework and tools are used to verify certain aspects of the control code for a Landing Gear System. The depth of the model was 5 and the number of reachable states in the order of 10 000, which is far less than the maximal 2^{26}.

Example 6.1 Reachable States
Consider again the interconnected system from Example 5.2 (Figure 5.7)

With $[0, 0]$ as initial state, we want to compute the reachable states. If $M(x, u_t, u_I, y_I, x^+)$ is the original relational model in the equation (5.6), then the relational model with only the transitions and no inputs or outputs can be written as

$$
\begin{aligned}
\tilde{M}(x, x^+) = \exists u_t, u_I, y_I.\ M = \\
& (x == [0,0]) \wedge (x^+ == [0,0]) \vee \\
& (x == [0,0]) \wedge (x^+ == [1,0]) \vee \\
& (x == [0,0]) \wedge (x^+ == [*,2]) \vee \\
& (x == [1,0]) \wedge (x^+ == [1,1]) \vee \\
& (x == [1,0]) \wedge (x^+ == [0,1]) \vee \\
& \qquad \vdots
\end{aligned}
\tag{6.3}
$$

where the continuation is easily seen from the figure. Since the model has the structure of Definition 5.5, we also have $\tilde{M}(x, x^+) = \mathsf{Project}\ x, x^+.\ M_t$, where M_t is from equation (5.4).

The set of reachable states are iteratively computed using (6.2) with $x == [0, 0]$ as initial state.

$$
\begin{aligned}
R_0(x) &= (x == [0,0]) \\
R_1(x) &= (x == [0,0]) \vee (x == [1,0]) \vee (x == [0,2]) \vee (x == [1,2]) \\
R_2(x) &= (x == [0,0]) \vee (x == [1,0]) \vee (x == [0,2]) \vee (x == [1,2]) \vee \\
&\quad (x == [1,1]) \vee (x == [0,1]) \\
R_3(x) &= R_2(x)
\end{aligned}
$$

Hence we reach a fixed point in $k = 2$ steps, and all $2 \cdot 3 = 6$ states are reachable in at most two steps.

Say instead that we are interested in the reachable states when the input u_t only takes the value 1. The relational model for this case is computed from the original relational model:

$$
\begin{aligned}
\tilde{M}(x, x^+) = \exists u_I, y_I.\ M[u_t \mapsto 1] = \\
&(x == [0,0]) \wedge (x^+ == [1,0]) \vee \\
&(x == [0,0]) \wedge (x^+ == [1,2]) \vee \\
&(x == [1,0]) \wedge (x^+ == [1,1]) \vee \\
&(x == [1,0]) \wedge (x^+ == [1,2]) \vee \\
&(x == [1,1]) \wedge (x^+ == [1,1]) \vee \\
&(x == [1,1]) \wedge (x^+ == [1,2]) \vee \\
&(x == [0,1]) \wedge (x^+ == [1,0]) \vee \\
&(x == [0,1]) \wedge (x^+ == [1,2]) \vee \\
&(x == [0,2]) \wedge (x^+ == [1,2]) \vee \\
&(x == [1,2]) \wedge (x^+ == [1,2])
\end{aligned}
\tag{6.4}
$$

If we again let the state $[0,0]$ be the initial state, the set of reachable states are iteratively computed using (6.2).

$$
\begin{aligned}
&R_0(x) = I(x) = (x == [0,0]) \\
&R_1(x) = (x == [0,0]) \vee (x == [1,0]) \vee (x == [1,2]) \\
&R_2(x) = (x == [0,0]) \vee (x == [1,0]) \vee (x == [1,2]) \vee (x == [1,1]) \\
&R_3(x) = R_2(x)
\end{aligned}
$$

Hence we again reach a fixed point in two steps, but only four states are reachable.
□

Another useful dynamic analysis is the computation of the backwards reachable states from a given, initial, state, i.e., the set of states from which the initial state can be reached. In this set, also the inputs needed to force the system to the initial state are included, regardless of the relational model being symmetric or not.

We will now define four generic reachable states operators.

Definition 6.1 Reachable States Operators
Let $M(z, z^+)$ be a relational model and let $I(z)$ represent the initial state set. For clarity, the arguments z and z^+ are not always given for the model and the initial state set.

We define $\gamma_k^+[M, I](z)$, $\gamma_k^-[M, I](z)$, $\Gamma_k^+[M, I](z)$ and $\Gamma_k^-[M, I](z)$ as follows:

Forward: Computes the set of forward reachable states $\gamma_k^+[M, I](z)$ from the set $I(z)$ in *exactly* k steps:

$$
\begin{aligned}
&\gamma_0^+[M, I](z) = I(z) \\
&\gamma_k^+[M, I](z) = \exists \tilde{z}.\ (\gamma_{k-1}^+[M, I](\tilde{z}) \wedge M(\tilde{z}, z))
\end{aligned}
\tag{6.5}
$$

Backward: Computes the set of backward reachable states $\gamma_k^-[M, I](z)$ from the set $I(z)$ in *exactly* k steps:

$$\begin{aligned}\gamma_0^-[M, I](z) &= I(z) \\ \gamma_k^-[M, I](z) &= \exists \tilde{z}.\ (\gamma_{k-1}^-[M, I](\tilde{z}) \wedge M(z, \tilde{z}))\end{aligned} \tag{6.6}$$

Accumulated Forward: Computes the set of forward reachable states $\Gamma_k^+[M, I](z)$ from the state set $I(z)$ in k *or less* steps. This is exactly the same formulation as in (6.2):

$$\begin{aligned}\Gamma_0^+[M, I](z) &= I(z) \\ \Gamma_k^+[M, I](z) &= \Gamma_{k-1}^+[M, I](z) \vee (\exists \tilde{z}.\ (\Gamma_{k-1}^+[M, I](\tilde{z}) \wedge M(\tilde{z}, z)))\end{aligned} \tag{6.7}$$

Accumulated Backward: Computes the set of backward reachable states $\Gamma_k^-[M, I](z)$ from the state set $I(z)$ in k *or less* steps:

$$\begin{aligned}\Gamma_0^-[M, I](z) &= I(z) \\ \Gamma_k^-[M, I](z) &= \Gamma_{k-1}^-[M, I](z) \vee (\exists \tilde{z}.\ (\Gamma_{k-1}^-[M, I](\tilde{z}) \wedge M(z, \tilde{z})))\end{aligned} \tag{6.8}$$

Remark: To compute the set of reachable states in any number of steps, $\Gamma_\infty^+[M, I](z)$, also denoted $\Gamma^+[M, I](z)$, we perform a fixed point computation, i.e.,

$$\Gamma_\infty^+[M, I](z) = \lim_{k \to \infty} \Gamma_k^+[M, I](z). \tag{6.9}$$

Since the set of accumulated reachable states is always increasing and we are dealing with a finite universe, a fixed point is reached for a finite k.

The set of backwards reachable states in any number of steps, $\Gamma_\infty^-[M, I](z)$, also denoted $\Gamma^-[M, I](z)$, is computed in the same manner. □

A useful analysis technique is the detection of loops, where a loop in a DEDS is a sequence of transitions that can repeat itself indefinitely. In Chapter 8, where diagnosis properties are analyzed, we will use this to find out if certain properties can hold forever and under which circumstances.

For a relational model $M(z, z^+)$, consider the fixed point computations

$$\gamma_\infty^+[M, I](z) = \lim_{k \to \infty} \gamma_k^+[M, I](z) \tag{6.10}$$

$$\gamma_\infty^-[M, I](z) = \lim_{k \to \infty} \gamma_k^-[M, I](z). \tag{6.11}$$

For the unaccumulated reachable states, convergence is not guaranteed in general, but if the initial states are *all* states, we have the following theorem.

Theorem 6.1 Unaccumulated Reachable States
The theorem consists of two parts.

- For a DEDS described by the relational model $\mathsf{M}(z, z^+)$, the following is true:

$$\gamma_0^+[\mathsf{M}, \mathtt{true}] \supseteq \gamma_1^+[\mathsf{M}, \mathtt{true}] \supseteq \dots \tag{6.12}$$
$$\gamma_0^-[\mathsf{M}, \mathtt{true}] \supseteq \gamma_1^-[\mathsf{M}, \mathtt{true}] \supseteq \dots \tag{6.13}$$

  From this it follows immediately that these computations will reach a fixed point such that

$$\gamma_\infty^+(z) = \gamma_k^+(z) = \gamma_{k+1}^+(z) \tag{6.14}$$
$$\gamma_\infty^-(z) = \gamma_j^-(z) = \gamma_{j+1}^+(z) \tag{6.15}$$

  for some finite k and j.

- The relation $\gamma_\infty^+[\mathsf{M}, \mathtt{true}](z)$, or $\gamma^+(z)$ for short, represents exactly the states that either belong to a loop or are forward reachable from a loop.

  The relation $\gamma_\infty^-[\mathsf{M}, \mathtt{true}](z)$, or $\gamma^-(z)$ for short, represents exactly the states and transitions that either belong to a loop or are backward reachable from a loop.

□

Proof The theorem will be proved by induction over the iterations.

Since the relation $\mathtt{true}$ represents all states of the system, $\gamma_0^+(z) = \mathtt{true}$ can be said to be *state invariant under* $\mathsf{M}(z, z^+)$, in the sense that M will not introduce any new states when used to compute the possible next states $\gamma_1^+(z)$ from $\gamma_0^+(z)$. That is, $\gamma_0^+(z) \supseteq \gamma_1^+(z)$.

The induction assumption is that the set of states $\gamma_k^+[\mathsf{M}, \mathtt{true}](z)$ is state invariant under M, i.e., $\gamma_k^+(z) \supseteq \gamma_{k+1}^+(z)$. The claim is that the set of possible next states $\gamma_{k+1}^+[\mathsf{M}, \mathtt{true}](z)$ also is state invariant under M. This follows from the induction assumption and the fact that the states that disappear are exactly those with no transitions into them from $\gamma_k^+(z)$ and hence from $\gamma_{k+1}^+(z)$. The first part of the theorem follows by induction. The analog induction works for $\gamma^-(z)$ also.

From the induction above, it is clear that the set $\gamma^+(z)$ contains exactly the states with transitions into them from $\gamma^+(z)$ itself. If this set is not empty, it can only consist of loops and the states forward reachable from the loops.

A similar argument goes for $\gamma^-(z)$. ■

Example 6.2 Loop detection
Figure 6.1 illustrates a case where we have two loops and an intermediate path between them. The dotted transitions and unfilled states are not included in the relation. □

Gunnarsson [30] introduces an operator that gives all loops and the intermediate paths in between loops for a given relational model $\mathsf{M}(z, z^+)$.

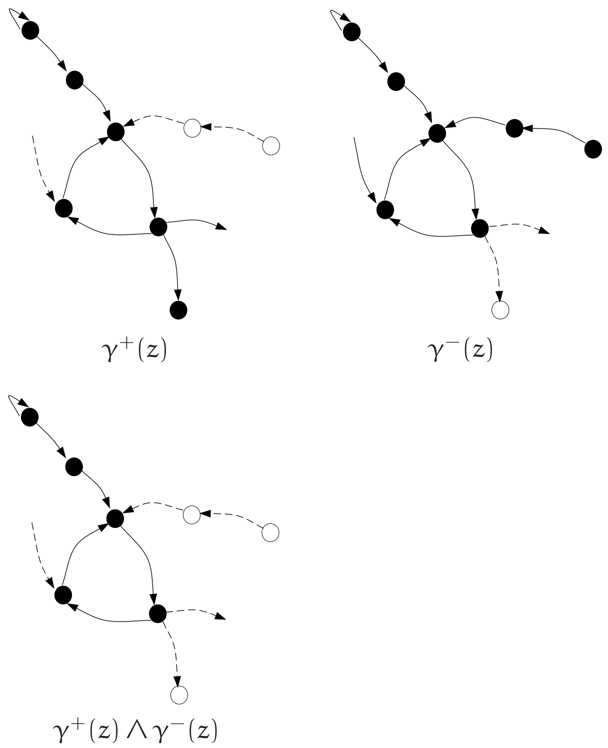


Figure 6.1: Unaccumulated reachable states forward and backward

Definition 6.2 Upper Approximation of Loops
The *upper approximation of loops*, denoted $\mathsf{ULoops}[M](z)$, is a relation which contains all states belonging to a loop or an intermediate path between loops of the system M. □

The computation of $\mathsf{ULoops}[M](z)$ is illustrated by the above example and figure 6.1. We formulate it as a lemma.

Lemma 6.2 Upper Approximation of Loops
For a relational model $M(z, z^+)$ we have

$$\mathsf{ULoops}[M](z) = \gamma^+[M, \mathtt{true}](z) \wedge \gamma^-[M, \mathtt{true}](z). \tag{6.16}$$

□

The lemma follows immediately from Theorem 6.1.

Chapter 7

Diagnosis

7.1 Introduction

With diagnosis we mean fault detection and fault isolation, i.e., we wish to determine from observations of the system if a fault has occured, where it occured and what it is. This rather loose statement will be given a more strict interpretation below.

In this chapter, we will assume that a relational model of the system is given

$$M(x, u, y, x^+) \tag{7.1}$$

where the states x, inputs u and outputs y can be vectors of variables,

$$\begin{aligned} x &= \{x_1, x_2, \ldots, x_{n_x}\} \\ u &= \{u_1, u_2, \ldots, u_{n_u}\} \\ y &= \{y_1, y_2, \ldots, y_{n_y}\} \end{aligned}$$

As mentioned in Section 5.2 and Section 5.4, each state variable models one component, as for the flow/pressure systems described in Section 5.8. As described in Section 5.9, faults are modeled with fault states in the value domain of the state (configuration) variables x. The value domains of the state variables are hence divided into *normal* and *faulty* values. From the framework, it follows that faults described by different fault states for the same variable are mutually exclusive. Any of the subsystems (components) can always, nondeterministically, choose to enter any fault state, and the fault is persistent in the model, i.e., the fault state can not be left once entered.

We will use the system model $M(x, u, y, x^+)$ divided into two parts, one for the next state transitions, and one for the behavior in each state

$$\begin{aligned} M_t(x, u, x^+) &= \mathsf{Project}\; x, u, x^+.\; M \\ M_{io}(x, u, y) &= \mathsf{Project}\; x, u, y.\; M \end{aligned} \tag{7.2}$$

according to Equation (5.1) following Definition 5.3.

With *nominal* state we mean the normal state the system is *supposed* to be in at a certain time step, according to some control scheme, i.e., choice of transition input u_t, together with the transition relation M_t (see Equation (7.4) below).

A system can be regarded as autonomous for the purpose of diagnosis, since we are only observing it. Hence, the relational model of the system does not have any acausal variables as in Definition 5.5 of component models. Such variables will generally be regarded as outputs, and usually represents sensors in the system.

In Section 7.2 and Section 7.3, fault detection and fault isolation given a relational model M will be discussed. Analysis of certain detection and isolation properties of the system model M are treated in Chapter 8.

7.2 Fault detection

Fault detection is basically about determining if something has gone wrong with a system using knowledge and observations of the system. A natural view to take is that if it is possible for a system to be in a fault free state, then it probably is. For the valve/pump system in Example 5.3, we would otherwise, for instance, warn that the valve may be stuck closed at every time step the valve is supposed to be closed. The following definition captures this view in terms of our DEDS framework.

Definition 7.1 Detection
Consider the relational model $M(x, u, y, x^+)$.

If at any time the known values of states, and the observed values of inputs and outputs together with M are not consistent with all state variables having a nominal value, then a fault has been detected.

Remark: Note that this definition says nothing about *which* fault or faults that have occured. This problem is called *fault isolation* and is treated in Section 7.3.
□

In the following, a procedure for obtaining fault detection for a system from observations and a relational model $M(x, u, y, x^+)$ will be outlined and explained.

At a given time step, we may have certain prior knowledge about the state of the system, in the form of a subset of the state variables definition domain X. We represent this prior system states with a relation $K_n(x)$, where n denotes the current time step. The prior knowledge usually is what system states the system could be in, according to some control scheme and the next state relation M_t. $K_n(x)$ is defined in Equation (7.4) below. Recall from Chapter 5 that faults are modeled as transitions to fault states, that a component can choose nondeterministically. Hence, the prior knowledge $K_n(x)$, in a certain time step, consists of the nominal state(s) and all possible component failures.

In the analysis part, Chapter 8, we will assume that there is exactly one nominal state, and that it is known, i.e., there is exactly one normal system state in $K_n(x)$.

At time step n, we also have observations of the inputs and outputs of the system, usually in the form of assignments to the variables u and y. Let these assignments be represented by the relations $U_n(u)$ and $Y_n(y)$. Together with the I/O part of the system model, $M_{io}(x, u, y)$, these assignments restrict the system states. The possible system states at time n, taking all observations into account, are then represented by the relation

$$D_n(x) = \mathsf{Project}\ x.\ M_{io}(x, u, y) \wedge K_n(x) \wedge U_n(u) \wedge Y_n(y). \tag{7.3}$$

Definition 7.2 Diagnosis
Consider a system model (7.2) and observed behavior from the real system (observations of the states, inputs and outputs).

We define a *diagnosis* as a complete state variable assignment consistent with the system model and the observed behavior, or more specifically as an element in the relation set of $D_n(x)$ from (7.3). □

The relation set of $D_n(x)$ will be called *the set of all diagnoses*, or estimation set.

The relation set of $D_n(x)$ represents all system states, or configurations, that are consistent with the systems observed behavior, and hence explains the behavior (w.r.t. to the system model). The structure of the diagnoses will be further investigated in Section 7.3. Here we only observe that a fault is detected according to Definition 7.1 if no normal system state is part of $D_n(x)$. The normal states that are not nominal are removed from D_n by the prior knowledge $K_n(x)$ in (7.3).

The possible next states to be used in the next iteration, $K_{n+1}(x)$ are given by the transition model $M_t(x, u, x^+)$, the now restricted possible current system states, $D_n(x)$, and the known values of the inputs:

$$K_{n+1}(x) = \mathsf{Project}\ x.\ M_t(\tilde{x}, u, x) \wedge D_n(\tilde{x}) \wedge U_n(u). \tag{7.4}$$

Algorithms are briefly discussed in Section 7.2.1.

7.2.1 Algorithms and complexity

The main aim of the detection method presented here is not implementation, but to show what can be done in principle, as a base for the analysis in Chapter 8. Nonetheless, we will make some remarks about the computational complexity, using notation, concepts and results from Chapter 4.

The IDDs representing the relations M_t, M_{io}, K_n and D_n will below be called F_t, F_{io}, F_k and F_d respectively. The inherent complexity for this kind of problems, "the state space explosion", primarily manifests itself as the worst case number of nodes of the IDDs F_t and F_{io}, which in both cases is exponential in the number of variables ($2n_x + n_u$ respectively $n_x + n_u + n_y$). In many practical cases, though, IDDs and BDDs are known to be significantly smaller than the worst case, which is the reason they are used, see further Chapter 4.

For a crude complexity analysis, the computation of D_n in (7.3) is divided in two parts; the restriction of the state variables with K_n and the assignment of values to the input and output variables. The restriction algorithm is worst case $\mathcal{O}(|\mathsf{F}_{io}||\mathsf{F}_k|)$, the assignment of variables is $\mathcal{O}(|\mathsf{F}_{io}|)$ (assignment of several variables simultaneously takes us closer to the worst case bound). The projection onto x is not necessary if the assignments are performed directly in the IDD graph, and it is recommended that the assignments, and hence removal of the corresponding variable, is performed first, since this empirically gives substantial space savings.

The algorithms can be made faster if we do not uphold the reducedness of the IDDs, but the worst case complexity is the same.

If all state variables are Boolean, with the two values interpreted as normal or faulty, we can check if the fault-free solution is part of the relation set of D_n in $\mathcal{O}(n_x)$, even if the BDD representing D_n is not reduced; there is exactly one path in the BDD that needs to be checked. If the state variables are not boolean and there are several normal states for each variable, the complexity is worse, but no worse than $\mathcal{O}(|\mathsf{F}_d|)$.

When the system (configuration) is controlled by a known, sequencing controller and the number of reachable states is comparatively low, an explicit state space representation, as in the formal language approach [70], could be favorable. It can be hard to find a good variable order for the IDD, and there are also relations for which *all* variable orders give an exponential size IDD-representation (see Chapter 4).

One way to make the classic tradeoff between speed and size, is to precompute the reachable D_n, as an explicit observer. Again, see [70].

7.3 Fault isolation

If some fault detection scheme has detected that the system is not working properly, the task of fault isolation ensues. For a repair action, the desired information is which component or components are faulty, and possibly what is wrong with them, depending on the level of detail and quality of the model. The definitions in this section are largely analogous to the AI model based diagnosis literature, see [32] for a comprehensive collection of articles.

From a practical point of view, we want to know which state variables, i.e., components, having a faulty value would explain the observed behavior. Then some form of repair action or other measures can be taken. We will now look into the structure of the set of diagnoses defined by Definition 7.2 and equation (7.3).

The set of all diagnoses might be quite large, since several different combinations of faulty components might be consistent with the behavior. In Example 7.1, we very informally use a classic example in the AI model based diagnosis literature, to motivate the following definition.

Example 7.1 The lightbulb-example
Given three parallel-coupled lightbulbs driven by a battery, we observe that no lightbulb is lit. Assuming the wiring is correct, and given what we know about electricity, the natural conclusion is that either the battery is dead, or all three

lightbulbs are broken. These are also consistent diagnoses. But also the battery dead together with any combination of broken lightbulbs are consistent diagnoses. This gives a set of totally 9 consistent diagnoses. □

Of practical interest, to decide on a repair action or further testing, would be to find the minimal diagnoses. A diagnosis is minimal if as few components as possible are declared faulty.

Definition 7.3 Explanations and Minimal diagnoses
Given is the set of all diagnoses (7.3).

The set of state variables (i.e., components) given faulty values by one diagnosis in the diagnoses set (7.3) is called an *explanation*.

The set of diagnoses (7.3) gives a set of explanations. A *minimal explanation* is an explanation that has no other explanation as subset. A diagnosis corresponding to a minimal explanation is called a *minimal diagnosis*. We call all minimal diagnoses the *minimal diagnosis set*, and all minimal explanations *the minimal explanation set*.

Remark:

- Since the subset-relation on the explanations is a partial order, there exists a set of minimal explanations/diagnoses for the diagnosis set (7.3), even though every explanation might correspond to several diagnoses (with different fault-values on the state variables).
- For convenience we will, with some abuse of notation, sometimes use the term "explanation" to refer also to a corresponding assignment of faulty values. When not stated explictly, this should be clear from context.

□

This definition of minimal diagnosis is basically an adaption to the non-boolean variable case from [20]. We will say that a diagnosis $\mathtt{d}_1$ is *greater than* a diagnosis $\mathtt{d}_2$, if the explanation set of $\mathtt{d}_1$ is a superset of the explanation set of $\mathtt{d}_2$. In the lightbulb-example 7.1, the minimal explanations are the two natural ones, {"battery dead"} and {"lightbulb 1 broken", "lightbulb 2 broken", "lightbulb 3 broken"}.

As noted in [20], an explanation, interpreted as an assignment, might very well cover system states that are not diagnoses, i.e., not members of the diagnosis set (7.3). In our opinion, the explanations are nonetheless a practically useful output from the diagnosis procedure.

The set of all minimal diagnoses can still be quite large, or at least costly to compute, see Section 7.3.1. A common focusing technique is the parsimony principle, since it usually is more probable that few components break simultaneously than many, especially if it can be assumed that components break independently of each other. A very common example of the parsimonity principle is the single fault assumption. A parsimonious minimal explanation set for the lightbulb example 7.1 consists of the single explanation {"battery dead"}.

One minimal diagnosis, the minimal diagnosis set, and all minimal diagnoses with up to a chosen number of faulty components can be computed from the IDD F representing the relation $D_n(x)$. See further Section 7.3.1.

7.3.1 Algorithms and complexity

In this section, we will outline algorithms that compute explicitly minimal diagnoses, Definition 7.3, from the set of all diagnoses (7.3) using the decision diagram data structure. None of the algorithms have been implemented yet. The set of all diagnoses is represented by a relation in the state (configuration) variables, here denoted $D(x)$, and the relation in turn is represented by an IDD F. IDDs and BDDs are introduced in Chapter 4, and the standard notation introduced there will be used.

For ease of notation, the state variables will be called $x_1, \dots, x_n$, where the numbering coincides with the variable order in the BDD (x_1 is topmost). Hence n is the number of variables. Using notation from Chapter 4, a vertex v represents a relation in the variables $x_{\mathsf{index}(v)}, \dots, x_n$. We will need to embed that relation in a larger space spanned by the variables $x_k, \dots, x_n$, where $k \leq \mathsf{index}(v)$. The embedded relation will be denoted $\mathsf{relation}(v, k)$. We use $\mathsf{component}(v)$ to indicate the component modeled by the state variable $x_{\mathsf{index}(v)}$.

There are two simple, but important observations fundamental for our algorithms computing minimal diagnoses from IDDs (BDDs). First, every vertex in the IDD F roots an IDD itself, in the variables $x_{\mathsf{index}(v)}, \dots, x_n$. Second, the minimal diagnoses for the relation represented by a vertex v, are given by combining the minimal diagnoses for the children of v (see the proof of Algorithm 7.2 for details).

One minimal diagnosis can be computed from the IDD F, as the lowest cost path from the top vertex to the true-terminal, where edges corresponding to fault-free state variable assignments are given weight zero, and the edges corresponding to faulty assignments are given weight one. This assignment is trivially a minimal diagnosis. We can e.g., use the well known Dijkstra algorithm, see e.g., [73], with worst case time complexity $\mathcal{O}(|F|)$, where $|F|$ is the number of vertices.

In general, breadth-first type algorithms are faster than depth-first algorithms, but the latter have smaller memory requirements, since they store only one path in memory at all times. For BDDs and IDDs, size is usually a topic and depth-first algorithms are normally used, see for example [7]. Below, the very basic depth-first Algorithm 7.1 for computing one minimal diagnosis, with the same worst case time complexity as Dijkstra, is presented. To keep the presentation of the algorithm as simple as possible, it is only presented in the Boolean case, but the extension to IDDs is straightforward.

Another alternative to depth-first and breadth-first strategies, would be to use a lattice algorithm, that explictly searches all paths with cost k before the paths with cost $k+1$. The worst case complexity would then be $\mathcal{O}(n|F|)$, since the graph might have to be traversed n times. The average performance would probably be superior, though, since the algorithm can terminate immediately when a diagnosis has been found. This is the search strategy employed in the General Diagnostic Engine (GDE) (Chapter 3 of [32]), although BDDs are not used there, but a set of propositional formulas together with a logic inference engine, to determine if a certain variable assignment is a diagnosis.

The minimal diagnosis set can also be computed from F with a depth first algorithm, as is demonstrated in Algorithm 7.2 in the Boolean case. This algorithm has worst case complexity $\mathcal{O}(|\mathsf{F}|n^2)$. The worst case number of traversed vertices is $\mathcal{O}(|\mathsf{F}|)$, but the operations in each vertex are worst case $\mathcal{O}(n^2)$ in the Boolean case (see remark after the proof of Algorithm 7.2).

The complexity for the operations in each vertex should on average be much better than $\mathcal{O}(n^2)$, but in order to find *all* minimal diagnoses, most of F needs to be traversed in general. This can be avoided by appealing to the parsimony principle and using a lattice algorithm. A nice feature of a lattice algorithm is that we can choose to stop after having computed all diagnoses with a certain number, k, of components faulty. By comparing with Algorithm 7.2, we get an upper limit on the worst case complexity for a lattice algorithm as $\mathcal{O}(k^3|\mathsf{F}|)$. The lattice algorithm needs at most k traversals of the graph, and the complexity for the operations in each vertex is k^2.

Better complexity measures can be found, at least for small k. For example, all single fault diagnoses, $k = 1$, are computed with the worst case complexity bounds $\mathcal{O}(n^2)$ and $\mathcal{O}(|\mathsf{F}|)$. The operations for each vertex are assumed constant time, since the number of pairwise comparisons of the childrens minimal diagnoses is exactly one (cf. Algorithm 7.2). The number of solutions the algorithm will check is $n+1$, and each check visits at most n vertices, hence $\mathcal{O}(n^2)$.

One minimal diagnosis algorithm

For simplicity of presentation, the algorithm is given in the Boolean case, i.e., it is assumed that the state variables are all Boolean and $x_i = 0$ means that the i:th component is normal, $x_i = 1$ means that the i:th component is faulty. The extension to IDDs is straightforward, since the value range for each state variable is partitioned in normal and faulty values (states). The algorithm (Algorithm 7.1) can certainly be optimized, for example by keeping track of the least cost so far, but is presented only to demonstrate the principle.

One minimal cost diagnosis, i.e., a minimal diagnosis, for each vertex in F is computed in a bottom-up fashion. The formulation is recursive, depth-first, and the terminals are assumed to have index $n + 1$, where n is the number of state variables. The algorithm is formulated using notation from Chapter 4. The global variables used in the algorithm are:

$\mathsf{visited}(v)$	A Boolean flag for each vertex.
$\mathsf{element}(v)$	The minimal diagnosis for the visited vertex v.
$\mathsf{cost}(v)$	The minimal diagnosis cost for the visited vertex v.

The local variables used in the algorithm are:

e, e_0, e_1	Intermediate diagnoses.
v_0, v_1	Inner vertices of the BDD.
$\mathsf{cost}_0, \mathsf{cost}_1$	Intermediate costs for diagnoses.

Initializations are left out to keep the formulation short.

Algorithm 7.1 One minimal diagnosis

The algorithm is in two parts, an "interface"-algorithm, $\mathsf{One}(F)$ called once, and a recursive part, $\mathsf{one}(v, e)$. The recursive part of the algorithm, $\mathsf{one}(v, e)$, is shown in Figure 7.1.

Syntax: $\mathsf{One}(F)$

Input: A BDD F, representing a relation in the state variables $x_1, \dots, x_n$.

Output: One minimal diagnosis for the relation, if any exists. Otherwise NULL.

Let v be the top vertex of F
if $\mathsf{one}(v, e) \neq \infty$ **then**
  Prepend e with $(\mathsf{index}(v) - 1)$ zeros
end if
return e

□

The algorithm has complexity $\mathcal{O}(|F|)$, since each vertex is visited at most once, and the operations for each vertex are constant time. The correctness proof for the similar, but more general Algorithm 7.2, is easily adapted.

One remark is that the purpose of the condition in line 22, Figure 7.1, is to make the algorithm work also for unreduced BDDs. This might be useful if speed really is of the essence. Our current IDD (BDD) implementation utilize the graph structure for operations as restriction and conjunction, so that the reducedness is maintained to a comparatively low cost, see Chapter 4 and [30, 7].

Minimal diagnosis set algorithm

Next, we present a minimal diagnosis set-algorithm. As the algorithm in the previous section, it is only given in the Boolean case. The idea in Algorithm 7.2 is to compute the minimal diagnosis set for all vertices in F starting at the bottom and merge the results upwards. The formulation is again recursive, depth-first.

The minimal diagnosis set for each vertex is represented explicitly, but the maximal number of minimal diagnoses for n Boolean state variables is n, so no state space explosion will occur. Also all vertices in F are not necessarily visited, so an upper limit on the worst case space requirement of the algorithm, apart from the BDD F itself, is $\mathcal{O}(|F|n)$.

Syntax: one(v, REF e)

Input: A BDD vertex v.

Output: One minimal diagnosis, e, for relation(v, index(v)), if it exists. Otherwise NULL. Returns the cost for that diagnosis, or ∞ if no diagnosis exists.

```
if value(v) == 1  then
  e = [ ]
  return 0
elsif value(v) == 0  then
  e = NULL
  return ∞
end if
if visited(v)  then
  e = element(v)
  return cost(v)
end if
visited(v) = 1

v_0 = child(v, 0), v_1 = child(v, 1)
cost_0 = one(child(v, 0), e_0)
if cost_0 > 1  then
  cost_1 = one(child(v, 1), e_1) + 1
else
  cost_1 = ∞, e1 = NULL
end if

if cost_0 == ∞  and cost_1 == ∞  then
  cost(v) = ∞, element(v) = NULL
elsif cost_1 < cost_0  then
  prepend e_1 with 1 followed by (index(v1) − index(v) − 1) zeros
  cost(v) = cost_1, element(v) = e1
else
  prepend e0 with (index(v0) − index(v)) zeros
  cost(v) = cost_0, element(v) = e0
end if
e = element(v)
return cost(v)
```

Figure 7.1: The recursive part of Algorithm 7.1 (one minimal diagnosis).

Global variables and data-structures used in the formulation of the algorithm are:

$\mathsf{visited}(\nu)$	A Boolean flag for each vertex.
$\mathsf{min\_diag\_set}(\nu)$	The computed minimal diagnosis set for each visited vertex ν.

The lookup table $\mathsf{min\_diag\_set}(\nu)$ is preferably hashed. The local variable used in the algorithm are:

S, S_0 and S_1	Intermediate sets of minimal diagnoses.
s_1, s_0	Intermediate diagnoses.
ν_0, ν_1	Inner vertices of the BDD.

The terminals are assumed to have the index $n+1$, where n is the number of state variables, and at line 13 in Figure 7.2, the empty diagnoses set $\{[\,]\}$ is interpreted as fault-free.

Algorithm 7.2 Minimal diagnosis set

The algorithm is in two parts, an "interface"-algorithm, $\mathsf{Mds}(F)$ called once, and a recursive part, $\mathsf{mds}(\nu)$. The recursive part of the algorithm, $\mathsf{mds}(\nu)$, is given in Figure 7.2.

Syntax: $\mathsf{Mds}(F)$

Input: A BDD F, representing a relation in the variables $x_1, \dots, x_n$.

Output: The minimal diagnosis set for the relation (possibly empty).

1 Let ν be the top vertex of F
2 $S = \mathsf{mds}(\nu)$
3 Prepend all elements in S with $(\mathsf{index}(\nu) - 1)$ zeros
4 **return** S

□

The following termination and correctness proof is quite trivial, and is given mainly to explain the algorithm.

Proof (Algorithm 7.2)

Termination is clear, since every vertex is visited at most once and the graph is finite.

If we show correctness of $\mathsf{mds}(\cdot)$, correctness for $\mathsf{Mds}(\cdot)$ follows trivially, since any other choice than 0 (correct) for the free variables (components) at line 3 per definition gives a non-minimal diagnosis. The proof idea is to use induction over the vertices from bottom to top. We will use notation introduced in the algorithm in the proof, and we also introduce $n_\nu = \mathsf{index}(\nu)$.

Syntax: mds(ν)

Input: A BDD vertex ν.

Output: The minimal diagnosis set for relation(ν, index(ν)).

```
if value(v) == 1 then
  return {[]}
elsif value(v) == 0 then
  return ∅
end if
if visited(v) then
  return min_diag_set(v)
end if
visited(v) = 1

v0 = child(v, 0), v1 = child(v, 1)
S0 = mds(v0)
if S0 consists of the fault-free diagnosis then
  S1 = ∅
else
  S1 = mds(v1)
end if

prepend (index(v0) − index(v) − 1) zeros to all elements in S0
prepend (index(v1) − index(v) − 1) zeros to all elements in S1
for s0 ∈ S0 do
  for s1 ∈ S1 do
    if s1 greater than s0 then
      remove s1 from S1
    end if
  end for
end for

prepend 0 to all elements in S0
prepend 1 to all elements in S1

min_diag_set(v) = S0 ∪ S1
return min_diag_set(v)
```

Figure 7.2: The recursive part of Algorithm 7.2 (minimal diagnosis set).

Initial step: The base for the induction is the two terminal vertices, here denoted T_0 and T_1. The boundary condition for the algorithm is that they are interpreted as relations in *no* variables, and hence $\mathsf{mds}(\mathsf{T}_1) = \{[\,]\}$, meaning that there *are* minimal diagnoses, and $\mathsf{mds}(\mathsf{T}_0) = \emptyset$, meaning that there are *no* diagnoses, are the correct return values. (This will be made more clear in the induction step below)

Induction step: Induction assumption: $\mathsf{mds}(\nu_0)$ is the minimal diagnosis set for $\mathsf{relation}(\nu_0,\ \mathsf{index}(\nu_0))$, and $\mathsf{mds}(\nu_1)$ is the the minimal diagnosis set for $\mathsf{relation}(\nu_1,\ \mathsf{index}(\nu 1))$.

The central claim is that we get exactly all minimal diagnoses for the relation represented by a specific vertex v from the two childrens minimal diagnoses sets.

Claim: At line 28, S_0 consists of exactly all minimal diagnoses for $\mathsf{relation}(\nu_0, n_\nu + 1)$.

The values of the "free" variables $x_{n_\nu+1}, \ldots, x_{\mathsf{index}(\nu_0)-1}$ do not, according to the BDD-definition, affect whether an assignment to the variables $x_{\mathsf{index}(\nu_0)}, \ldots, x_n$ is part of a diagnosis[1]. The only minimal choice for the free variables is trivially all corresponding components fault-free, i.e., $x_{n_\nu+1} = \ldots = x_{\mathsf{index}(\nu_0)-1} = 0$, since any other choice would give a non-minimal diagnosis (Definition 7.3). Together with the induction assumption, this proves the claim.

Claim: At line 28, S_1 consists exactly of the minimal diagnoses for $\mathsf{relation}(\nu_1, n_\nu + 1)$, minus the diagnoses that are greater than a diagnosis in S_0 at line 28.

Without the condition at line 13 and the loop at line 21, S_1 would consist of exactly all minimal diagnoses for $\mathsf{relation}(\nu_1,\ n_\nu + 1)$. Same proof as for S_0. From this set, all diagnoses that are greater than or equal to a diagnosis in S_0, are removed by the loop at line 21. The condition at line 13 only handles a special case, since any diagnosis is equal to or greater than the fault-free diagnosis. The claim is proved.

At line 31, the diagnoses in S_1 and S_0 have been prepended with the correct assignment on $\mathsf{component}(\nu)$. Since $\mathsf{component}(\nu)$ must be either correct or faulty in a diagnosis, the two claims give that $S_0 \cup S_1$ contains all minimal diagnoses for $\mathsf{relation}(\nu,\ n_\nu)$.

$S_0 \cup S_1$ also contains only minimal diagnoses, since all diagnoses in S_1 greater than a diagnosis in S_0 has been removed, and a diagnosis in S_0 cannot be greater than a diagnosis in S_1, since all diagnoses in S_1 has $\mathsf{component}(\nu)$ faulty, and $\mathsf{component}(\nu)$ is correct for all diagnoses in S_0.

Hence the algorithm is correct.

■

Note that the structure of a reduced BDD is not used in the proof, hence the algorithm works also on unreduced BDDs.

To make a simple complexity analysis of the algorithm, assume that table-lookups are made in constant time. Each vertex in the BDD F is visited at most once. For a specific vertex ν, let N_0 and N_1 be the number of minimal diagnoses for the child vertices, ν_0 and ν_1. The prepending after lines 18 and 28 are $\mathcal{O}(N_0 + N_1)$.

[1] There are free variables if $n_\nu + 1 \leq \mathsf{index}(\nu_0) - 1$

The removal of non-minimal diagnoses for $\mathsf{relation}(\nu,\ \mathsf{index}(\nu))$ at 21 is $\mathcal{O}(N_0 N_1)$. Since the maximal number of minimal diagnoses given n (Boolean) variables, is n, the worst case complexity for the whole algorithm is $\mathcal{O}(|F| n^2)$. Of course, worst case for the number of vertices is $|F| = 2^n$.

Chapter 8

Analysis of diagnosis properties

The analysis in this chapter is basically an investigation of *if* and *when* faults can be detected/isolated in a system described by a relational model as in Section 7.1.

A basic assumption on which the analysis rests is that the nominal state of the system is exactly known and that this knowledge is utilized in the diagnosis procedure. In the terms of the previous chapter, we assume that $\mathsf{D}_\mathsf{n}(\mathsf{x})$, Equation (7.3), contains exactly one fault-free system state at all times.

We hence assume that the nominal behavior of the system described by the relational model M, is deterministically observable (see Section 5.3). Note that this assumption is only for this chapter, it is not necessary for the detection and isolation methods in Chapter 7.

Most of the detectability and isolatability analysis presented here is static, and is performed on the pure I/O behavior, restricted to the feasible, or reachable states, assuming that the nominal state is known and utilized in the diagnosis. In Section 8.1 dynamic analysis is performed regarding whether faults can stay undetected infinitely, which is very much a property of the controller scheme used for the system.

One serious restriction for the analysis methods in Section 8.1 and Section 8.2 is the following assumption on the transitions of the system.

Assumption 8.1 *A fault in one component does not influence the state evolution of another component.*

In terms of the DEDS component model, Definition 5.5, no component output that is affected by faults in the component model is allowed to be connected to a transition input of another component model. The analysis is much simplified if the state of the non-faulty components can be used to keep track of the nominal system state, and we can perform an investigation of detection and isolation properties for the whole state space with a few relational operations.

8.1 Analysis of detection properties

The very simple idea is to pick out the nominal behavior for which the fault is *not* detected. The analysis is carried out separately for each fault mode of each component, under a single fault assumption. That is, we analyse detectability properties under the assumption that only one fault has occurred, with the rest of the components performing nominally.

We will use the same setup and notation for the model and the variables as in Chapter 7. The individual fault values for each state variable x_i that has fault values, are denoted $f_{i,k}$, where k identifies the individual fault. To check detectability for a specific fault, we will use the relational model where all components except the faulty one is assumed correct:

$$M_i(x, u, y, x^+) = \mathsf{Restrict}[M, N_i]. \tag{8.1}$$

respectively

$$\begin{aligned} M_{t,i}(x, u, x^+) &= \mathsf{Restrict}[M_t, N_i] \\ M_{io,i}(x, u, y) &= M_{io} \wedge N_i \end{aligned} \tag{8.2}$$

The operator Restrict is introduced in Definition 5.7 in Section 5.7. The relations $N_i = N_i(x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_{n_x}), 1 \leq i \leq n_x$ restricts all state variables except x_i to their normal values. We also let M_N respectively $M_{t,N}, M_{io,N}$ be the relational models for the whole system restricted to normal behavior.

For all analysis we also assume that M has been restricted to its reachable states.

$$M = \mathsf{Restrict}[M, \Gamma^+[M, I]]. \tag{8.3}$$

Hence if a controller scheme for the system is included in M_t, then $M_N, M_{io,N}$ and $M_{t,N}$ represent the reachable normal behavior of the system. For systems modeled e.g., with the methods in Section 5.8, the I/O relation M_{io} for a complete system will not have any inputs at all, but all external inputs to the system are pure transition inputs, i.e., then $M_{io}(x, u_{io}, y) = M_{io}(x, y)$ and $M(x, u, y, x^+) = M(x, u_t, y, x^+)$.

Definition 8.1 Detectable fault
A fault $f_{i,k}$ is *detectable* if there exist feasible (reachable) normal states and inputs in $M_{io}(x, u_{io}, y)$ for which the fault is detected (Definition 7.1), and there exists controllable transitions in $M_t(x, u_t, x^+)$ that will lead to such a state in a finite number of steps. □

The transitions that lead to detection correspond to the indicator events in the formal language approach by Sampath et al. [70]. We also say that M is *detectable* if all faults are detectable. This definition of detectability then corresponds to I-diagnosability defined in [70], with the partition of the faults discussed in Section 8.2. Note that detectable fault and detectable are properties of the relational model M and not of the real system.

In the following, algorithmic methods will be presented to decide if a fault is detectable (Section 8.1.1), and to decide which transitions and hence which transition inputs that are necessary to detect the fault (Section 8.1.2).

8.1.1 Ambivalent behavior

Due to the Assumption 8.1 that the fault does not affect any other components state transitions, the difference in I/O behavior is the only observable difference between a nominal and a faulty system. At each time step, we will say that a nominal system state x and input u_{io} (input of M_{io}) are *ambivalent w.r.t.* $f_{i,k}$, if we by observing the outputs y cannot distinguish the state x from the system state where all variables have the same values except for $\mathsf{x}_\mathsf{i} = f_{i,k}$.

Hence, under a single fault assumption on the real system, the fault $f_{i,k}$ cannot be detected according to Definition 7.1, even though D_n from the Equation (7.3) contains x as only normal state.

We will also say that the whole fault detection procedure in Section 7.2 is ambivalent w.r.t. $f_{i,k}$ when both the nominal state and a system state where $\mathsf{x}_\mathsf{i} = f_{i,k}$ is in the estimation set $\mathsf{D}_\mathsf{n}(\mathsf{d})$ simultaneously, i.e., when the faulty *and* the nominal behavior is consistent with the observations.

Ambivalence is a property of the model, but it might also be an inherent property of the real system, that no modeling in the world can change. For example, for flow/pressure systems as in Section 5.8 and 5.9, at least one possible fault is bound to be in the estimation set D_n at almost all times. When a valve is closed there is always the possibility that it is stuck closed, and we cannot know that without trying to open it.

The ambivalent nominal I/O behavior for the fault $f_{i,k}$ can be computed from the I/O part $\mathsf{M}_{\mathrm{io},\mathsf{i}}$ of the system relation,

$$\mathsf{A}_{f_{i,k}}(\mathsf{x}, \mathsf{u}_{\mathrm{io}}, \mathsf{y}) = \mathsf{M}_{\mathrm{io},\mathsf{i}}[\mathsf{x}_\mathsf{i} \mapsto f_{i,k}] \wedge \mathsf{M}_{\mathrm{io},\mathsf{N}}. \tag{8.4}$$

$\mathsf{M}_{\mathrm{io},\mathsf{N}}(\mathsf{x}, \mathsf{u}_{\mathrm{io}}, \mathsf{y})$ is the I/O behavior for all nominal system states. The relational model

$$\mathsf{M}_{\mathrm{io},\mathsf{i}}[\mathsf{x}_\mathsf{i} \mapsto f_{i,k}]([\mathsf{x}_1, \dots, \mathsf{x}_{\mathsf{i}-1}, \mathsf{x}_{\mathsf{i}+1}, \dots, \mathsf{x}_{\mathsf{n}_\mathsf{x}}], \mathsf{u}_{\mathrm{io}}, \mathsf{y})$$

is the I/O behavior for the system states where all variables have the same values as for a nominal state, except for x_i.

$\tilde{\mathsf{A}}_{f_{i,k}}(\mathsf{x}, \mathsf{u}_{\mathrm{io}}) = \exists \mathsf{y}.\ \mathsf{A}_{f_{i,k}}$ is the nominal system states (and inputs) where the observed output does not distinguish between the real system being nominal or faulty. Note that we consider the ambivalence for each state/input combination simultaneously, and this is hence an example of static analysis, made possible by Assumption 8.1 that the fault does not influence the state transitions of the rest of the system.

If the transition behavior M_t of the system incorporates a control scheme that sequences system states, we would also like to know if it is possible to have a sequence of states such that we never can detect $f_{i,k}$. That is, we want to find the loops in $M_i(x, u, y, x^+)$ where the fault detection procedure is ambivalent w.r.t. $f_{i,k}$. These loops will be called the *infinitely ambivalent behavior*.

To find this behavior, we restrict $M_N(x, u, y, x^+)$ with the ambivalent nominal I/O behavior given by $A_{f_{i,k}}(x, u_{io}, y)$.

$$M_{f_{i,k}}(x, u, y, x^+) = M_N(x, u, y, x^+) \wedge A_{f_{i,k}}(x, u_{io}, y). \tag{8.5}$$

The relational model $M_{f_{i,k}}$ now represents exactly the nominal behavior in M for which the detection procedure in Section 7.2 is ambivalent w.r.t. the fault $f_{i,k}$.

To find the behavior such that the fault $f_{i,k}$ never is detected, we need to find the loops of $M_{f_{i,k}}(x, u, y, x^+)$ (see Section 6.2). Loops in a relational model are detected by the operator $\mathsf{ULoops}[\,\cdot\,]$ introduced in Definition 6.2. If we compute

$$R(x, u, y) = \mathsf{ULoops}[M_{f_{i,k}}](x, u, y)$$

then the infinitely ambivalent behavior of M w.r.t. $f_{i,k}$ is given by

$$M^{\infty}_{f_{i,k}}(x, u, y, x^+) = \mathsf{Restrict}[M_{f_{i,k}}, R](x, u, y, x^+). \tag{8.6}$$

The operator Restrict is introduced in Definition 5.7 in Section 5.7. The procedure is illustrated in Figure 8.1.

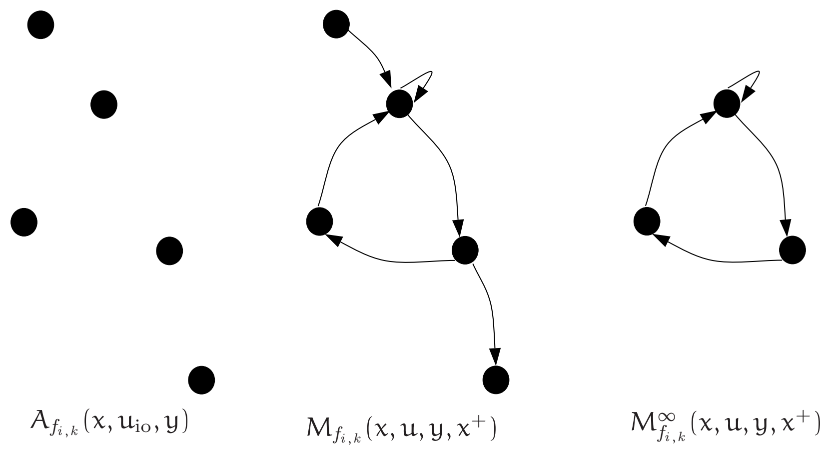


Figure 8.1: Finding the infinitely ambivalent behavior.

The nominal transition behavior for which the the detection procedure is ambivalent is contained also in

$$M_{t,f_{i,k}}(x, u_t, x^+) = \exists y.\ M_{f_{i,k}}(x, u, y, x^+) \tag{8.7}$$

As mentioned above, a complete system model may very well have only pure transition inputs, as is the case for systems modeled with the methods in Section 5.8.

In that case, we can call a nominal state ambivalent without taking the (I/O) input into account.

To analyze the infinite ambivalent behavior for a specfic system and specific fault, we can, e.g., generate a sample path of $\mathsf{M}^{\infty}_{f_{i,k}}(x, u, y, x^{+})$ using random simulation, or study specifically the transitions that lead out of this behavior, see Section 8.1.2.

If it is feasible, we can also visualize for example $\mathsf{M}^{\infty}_{t,f_{i,k}}(x, u_t, x^{+})$ as a FSM (see Example 8.1).

An alternative to using ULoops, is to only do a backwards reachability computation when computing $\mathsf{M}^{\infty}_{d_{i,k}}$, leaving also the ambivalent behavior that leads *into* the loops.

8.1.2 Detectability test

The faults that never can be detected regardless of input (transition and ordinary), are called *non-detectable* according to Definition 8.1 and are of course of special interest. A test for the detectability of the fault $f_{i,k}$ is simply a comparison of the full nominal behavior of the system and the nominal behavior for which the fault cannot be detected

$$\mathsf{M}_{\mathsf{N}}(x, u, y, x^{+}) \stackrel{?}{=} \mathsf{M}_{f_{i,k}}(x, u, y, x^{+}). \tag{8.8}$$

If this test comes out true, then $f_{i,k}$ is non-detectable. That M_{N} covers $\mathsf{M}_{f_{i,k}}$ is clear from construction.

If we focus on the transition behavior, the transitions that lead out of $\mathsf{M}^{\infty}_{t,f_{i,k}}$, and hence will lead to a detection in a finite number of steps, are explicitly given by the relation

$$\mathsf{M}_{t,\mathsf{N}}(x, u_t, x^{+}) \wedge \neg \mathsf{M}^{\infty}_{t,f_{i,k}}(x, u_t, x^{+}) \wedge \mathsf{Project}\ x.\ \mathsf{M}^{\infty}_{t,f_{i,k}} \tag{8.9}$$

The first two relations together give all transitions not in the infinitely ambivalent transition behavior. The third relation restricts the transitions to the ones originating from the infinitely ambivalent behavior. The so found transitions then correspond to the indicator events in the formal language approach [70].

Example 8.1 Valve/Pump example revisited

We will now analyse the valve/pump system modeled in Example 5.3, Section 5.10, with the methods discussed in this section.

The faults that can occur in the valve will be denoted $f_{v,\mathrm{SC}} = \mathrm{SC}$ for valve stuck closed and $f_{v,\mathrm{SO}} = \mathrm{SO}$ for valve stuck open. The nominal transition behavior is shown as a FSM in Figure 8.2[1]. The states in the figure are denoted with the

[1] All visualizations of FSMs are generated from the IDD-representation using Alon Levy's FiniteAutomata package for *Mathematica* [81].

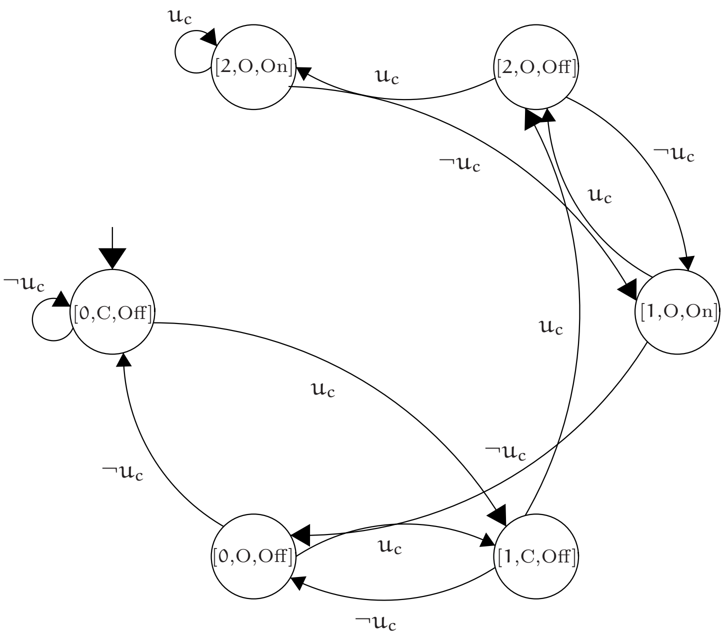


Figure 8.2: Nominal behavior for the valve/pump system in Example 8.1.

corresponding element from the state space $X_c \times X_v \times X_p$, where

$$\begin{aligned} X_c &= \{0, 1, 2\} \\ X_v &= \{\mathrm{O}, \mathrm{C}, \mathrm{SO}, \mathrm{SC}\} \\ X_p &= \{\mathrm{On}, \mathrm{Off}, \mathrm{SOn}, \mathrm{SOff}\} \end{aligned}$$

as is defined in Example 5.3. The value of the transition input u_c is denoted as a boolean literal in the figure, i.e., u_c means $u_c == 1$ and $\neg u_c$ means $u_c == 0$. The ambivalent behavior for the valve stuck closed $M_{t,f_{v,\mathrm{SC}}}$ computed according to Equation (8.5) and Equation (8.7) is shown in Figure 8.3 (there is no ordinary input). In the infinitely ambivalent behavior $M^{\infty}_{f_{v,\mathrm{SC}}}$ from Equation (8.6) the state [2,O,Off] has been removed, as is shown in Figure 8.4. As is easily seen, the fault is detectable, and the single indicator transition that will lead to detection, according to Equation (8.9), is shown in Figure 8.5.

This means that if flow is commanded, $u_c = 1$, the valve is supposed to open and the pump to start pumping and we should get flow. If the valve is stuck closed, though, we do not get any flow and the fault $f_{v,\mathrm{SC}}$ can be detected.

If we compute the ambivalent behavior $M_{f_{v,\mathrm{SO}}}$, it turns out to be identical to

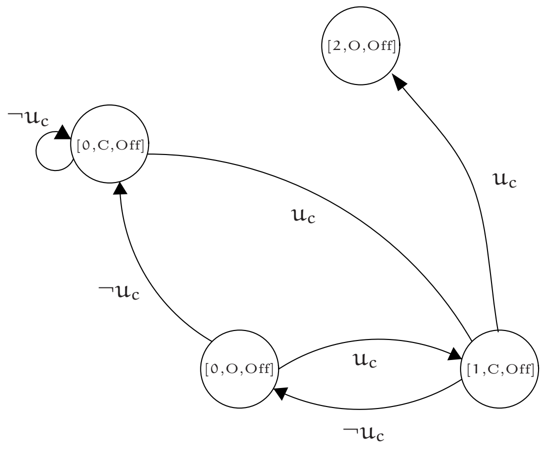


Figure 8.3: The ambivalent behavior $M_{f_v,\mathrm{SC}}$ for the valve/pump system in Example 8.1.

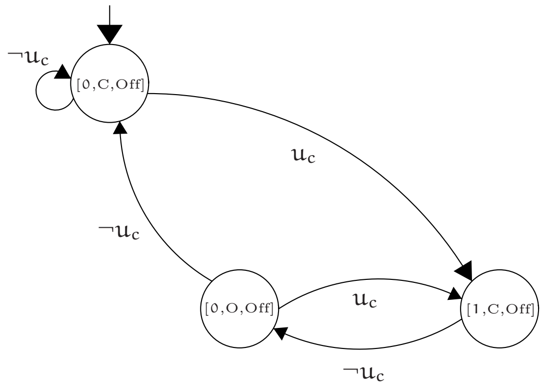


Figure 8.4: The infinitely ambivalent behavior $M^{\infty}_{f_v,\mathrm{SC}}$ for the valve/pump system in Example 8.1.

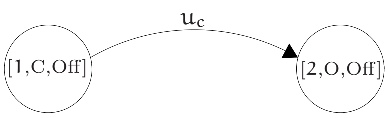


Figure 8.5: The indicator transition for the fault $f_{v,\mathrm{SC}}$ in the valve/pump system in Example 8.1.

the nominal behavior M_N visualized in Figure 8.2. Hence the test (8.8) comes out true and the fault $f_{v,\mathrm{SO}}$ is non-detectable.

The explanation is that to detect the valve stuck open by flow measurements, we would have to run the pump when the valve is supposed to be closed, and the controller never does that. Here we see that even if there exist "open-loop" system configurations where a fault is detected (normal state $x_v = \mathrm{C}, x_p = \mathrm{On}$), these states might be excluded from the feasible states by the controller scheme. In this case, it is for safety reasons. □

8.2 Analysis of isolation properties

After a fault has been detected, i.e., when a fault-free system state is no longer in the estimation set D_N from Equation (7.3), we might have a number of minimal diagnoses containing one faulty component.

The property we are after, under the single-fault assumption on the real system, is which faults can be distinguished from each other, that is, which faults are not always simultaneously in D_n (as minimal diagnoses). An isolatability property can be defined in many ways, depending, e.g., on what we are prepared to do with the system to achieve isolation, and if we demand symmetry in the isolatability between two faults.

We will limit our analysis to two simple investigations: whether is is theoretically possible to distinguish two faults from each other, and which faults can be immediately isolated upon detection.

If two faults shall be possible to distinguish at all, the systems behavior have to differ in some respect when they have occured. Under Assumption 8.1, that a fault do not affect the state transitions of other components, it is sufficent to compare the faulty I/O behavior for all states and inputs of the system. For two faults $f_{i,k}$ and $f_{j,l}$, we form the faulty I/O behavior relations

$$\begin{aligned} \mathsf{R}_{f_{i,k}}(x,u,y) &= M_{\mathrm{io},i}[x_i \mapsto f_{i,k}] \wedge \tilde{M}_{\mathrm{io},N} \\ \mathsf{R}_{f_{j,l}}(x,u,y) &= M_{\mathrm{io},j}[x_j \mapsto f_{j,l}] \wedge \tilde{M}_{\mathrm{io},N} \end{aligned} \tag{8.10}$$

where $\tilde{M}_{\mathrm{io},N}(x,u) = \exists y.\, M_{\mathrm{io},N}$. The only difference from the ambivalent behavior calculation in equation (8.4), is that we do not intersect the faulty I/O behavior with the nominal I/O behavior, due to the quantification of y in $\tilde{M}_{\mathrm{io},N}(x,u)$. Hence, we get all the reachable normal system states, but with I/O behavior as if the fault $f_{i,k}$ has occured. No information is lost by restricting the system states to normal states, due to Assumption 8.1. The reason for superimposing the faulty I/O behavior onto the normal system states in Equation (8.10), is so the two relations can be easily compared.

Definition 8.2 Isolatable faults
The two faults $f_{i,k}$ and $f_{j,l}$ are *isolatable* iff their respective I/O behavior are different. That is, iff

$$\mathsf{R}_{f_{i,k}} \neq \mathsf{R}_{f_{j,l}}$$

holds. □

That two faults give different I/O behavior does not mean that any of them are detectable (Definition 8.1). The property is that if a fault has been detected according to Equation (7.3), i.e., the fault free system state is no longer in D_n, then we can distinguish between the two faults for some system state.

The only difference between the faulty I/O behavior relation $\mathsf{R}_{f_{i,k}}$ and the ambivalent behavior relation $\mathsf{A}_{f_{i,k}}$, is that the faulty I/O behavior relation is intersected with a relation with larger relation set. That is, $\tilde{\mathsf{M}}_{\mathrm{io},\mathsf{N}}$ instead of $\mathsf{M}_{\mathrm{io},\mathsf{N}}$. Hence a sufficient condition for the faults to be isolatable is

$$\mathsf{A}_{f_{i,k}}(\mathsf{x},\mathsf{u},\mathsf{y}) \neq \mathsf{A}_{f_{j,l}}(\mathsf{x},\mathsf{u},\mathsf{y}).$$

Hence, if this test shows that the faults are isolatable, the faulty I/O behavior relations in Equation (8.10) need not be computed for this comparison.

To find all pairs of faults that cannot be distinguished, we have to perform all possible pairwise comparisons. Since the analysis is done off line, this should be reasonable in many cases. What we get is a list of pairs of faults that cannot be isolated without, e.g., changing the controller or adding sensors.

The finest possible fault partition, under the single fault assumption, for which the faults in different partitions are isolatable, is then given by partitioning the faults with identical I/O behavior together.

Two faults $f_{i,k}$ and $f_{j,l}$ are called *immediately isolatable* if it is guaranteed that only one of them is part of D_n as a minimal diagnoses when a fault is first detected (Definition 7.1). This is a stricter property than isolatable, i.e., if two faults are immediately isolatable, they are also isolatable.

The nominal states $\tilde{\mathsf{M}}_{\mathrm{io},\mathsf{N}}(\mathsf{x},\mathsf{u}_{\mathrm{io}})$ are divided into four separate sets by the ambivalent sets $\tilde{\mathsf{A}}_{f_{i,k}}(\mathsf{x},\mathsf{u}_{\mathrm{io}})$ and $\tilde{\mathsf{A}}_{f_{j,l}}(\mathsf{x},\mathsf{u}_{\mathrm{io}})$. To check for immediate isolatability, we need only compare the faulty I/O behavior where both faults can be detected. Here, though, the I/O behavior must be different for all states. The states where both faults are detectable are the intersection of the non-ambivalent states, i.e., the intersection of the complements of $\tilde{\mathsf{A}}_{f_{i,k}}$ and $\tilde{\mathsf{A}}_{f_{j,l}}$ w.r.t. $\tilde{\mathsf{M}}_{\mathrm{io},\mathsf{N}}$:

$$\mathcal{D}(\mathsf{x},\mathsf{u}) = \neg\tilde{\mathsf{A}}_{f_{i,k}}(\mathsf{x},\mathsf{u}) \wedge \neg\tilde{\mathsf{A}}_{f_{j,l}}(\mathsf{x},\mathsf{u}) \wedge \tilde{\mathsf{M}}_{\mathrm{io},\mathsf{N}}(\mathsf{x},\mathsf{u}).$$

To check if the two faults are immediately isolatable, we now need only check if

$$\mathsf{R}_{f_{i,k}}(\mathsf{x},\mathsf{u},\mathsf{y}) \wedge \mathsf{R}_{f_{j,l}}(\mathsf{x},\mathsf{u},\mathsf{y}) \wedge \mathcal{D}(\mathsf{x},\mathsf{u}) = \mathtt{false}.$$

Two faults can be isolatable but not immediately isolatable if, e.g., the output of the system is not specified in all system states, as is the case for pressure in the systems discussed in Section 5.8.

Example 8.2 Multiple faults

Once again, we will analyse the valve/pump system modeled in Example 5.3, Section 5.10.

The faults in the valve are the same as in Example 8.1 and the pump faults will be denoted $f_{p,\mathrm{SOff}} = \mathrm{SOff}$ for pump stuck off and $f_{p,\mathrm{SOn}} = \mathrm{SOn}$ for pump stuck on.

When analyzing the detection properties, we discover that both pump stuck off $f_{p,\mathrm{SOff}}$ and pump stuck on $f_{p,\mathrm{SOn}}$ are detectable, see Figure 8.6. The nominal behavior to compare with is in Figure 8.2.

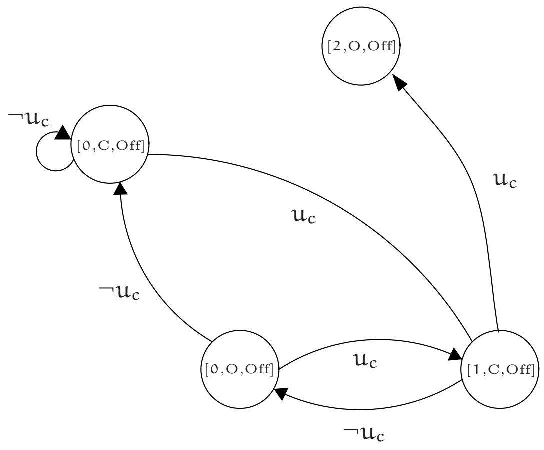
When we follow the procedure outlined in this section to decide which faults are pairwise isolatable, it turns out that $f_{v,\mathrm{SC}}$ and $f_{p,\mathrm{SOff}}$, i.e., valve stuck closed and pump stuck off, cannot be distinguished. In either case, we simply do not get any flow at the flow sensor.

Hence, if we introduce the fault partition

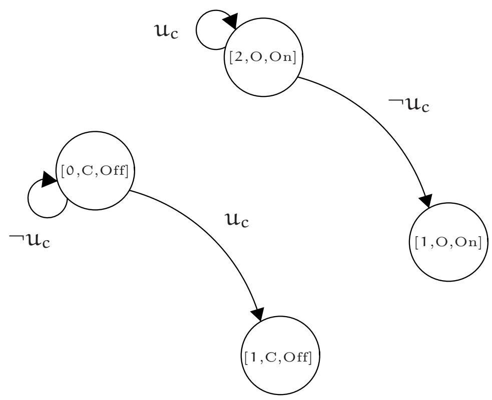
$$\{\{f_{v,\mathrm{SC}},\ f_{p,\mathrm{SOff}}\},\ \{f_{v,\mathrm{SO}}\},\ \{f_{p,\mathrm{SOn}}\}\} \tag{8.11}$$

these groups of faults are isolatable from each other, apart from that $f_{v,\mathrm{SO}}$ is undetectable. We get the same partition for the immediately isolatable property.

A sufficient condition for two faults to be isolatable is of course that the behavior for which the two faults stay undetected are different. Therefore the above partition is hinted already if we compare Figure 8.3 and 8.6. □



(a) The nominal system states for which $f_{p,\mathrm{SOff}}$ (pump stuck off) cannot be detected.



(b) The nominal system states for which $f_{p,\mathrm{SOn}}$ (pump stuck on) cannot be detected.

Figure 8.6: The ambivalent behavior w.r.t. $f_{p,\mathrm{SOff}}$ and $f_{p,\mathrm{SOn}}$.

Chapter 9

Summary and Conclusions, Part I

We have proposed an approach for modeling and diagnosis of systems that fall in the area of discrete event dynamic systems and shown how to perform analysis of diagnosability properties, and also shown on the tight relationship with model based diagnosis in AI. We have presented automatic procedures for finding which faults are detectable, the necessary behavior to detect a fault, and also how to find the finest possible fault partition.

The most important future work is the continuing validation of the approach to DEDS and diagnosis of DEDS presented in this thesis.

The practical implications of the difference in synchronous product between the formal language, RW, approach and the relational approach mentioned in Section 5.5 need to be further investigated. Also, the diagnosis analysis need to be extended to allow nondeterminism not emanating from faults and faults that affect state transitions of other components.

Our current computer tools for representation and computation are well suited to perform the analysis treated in Chapter 8 for large systems, but the online real-time fault detection proposed in Section 7.3.1 is not yet implemented, and also needs to be benchmarked against existing tools. The interface to the tools are currently also somewhat primitive and a more sophisticated modeling environment and modeling language is needed.

The above remarks can be summarized in the need to apply the framework to a larger, real diagnosis problem and learn from the experience.

Part II

Fault handling in object oriented control systems

Chapter 10

Introduction Part II

Developing control systems for complex systems is a difficult and increasingly important task. Larger control systems have traditionally been developed using structured analysis and functional decomposition (see e.g. DeMarco [21]). Today, many large systems are designed using an object oriented (OO) approach, see e.g., [22, 66, 74]. This has several advantages over traditional approaches, including better possibility to cope with complexity and to achieve good maintainability and reusability (see e.g. [6]). It leads, though, to new kinds of problems, and in Part II of this thesis, we will concern ourselves with the problem of fault propagation caused by an object oriented software architecture.

Object oriented design goals such as *encapsulation* and *modularity* often stand in direct conflict with the need to generate concise information about a fault scenario, and to avoid propagating error messages. Error messages are sent by individual objects to notify, e.g., an operator that the object has detected an error condition. The OO goal to encapsulate information implies that individual objects or group of objects do not in general know how close they are to the fault, or if the fault has already been adequately reported, and hence whether they should log an error message or not.

When a fault situation occurs, e.g., a hardware component failure, a broken communication link or a real-time fault, it is important that the control system generates clear and concise information about the occured fault to an operator, so that it can be quickly repaired and normal operation restored. To present the operator with a multitude of error messages from different parts of the system are hence not a very desirable behavior. The problem from the operators point of view is schematically illustrated in Figure 10.1.

Using traditional software development methods, it is formally possible to have the state of the whole control system known centrally. It is hence possible, at least formally, to generate concise information to an operator about a fault condition. In object oriented design, encapsulation and modularity are fundamental and important design goals for reuse, maintenance and complexity reasons. It will be further motivated in Section 10.1 how these object-oriented design goals often

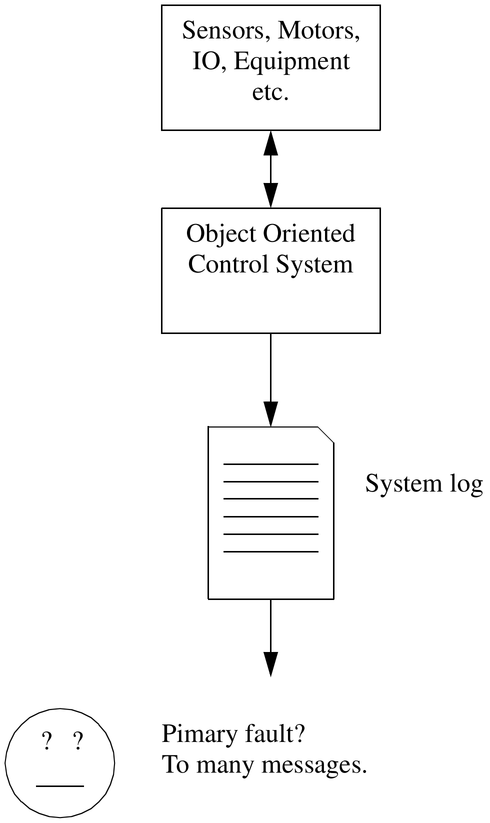


Figure 10.1: Problem overview

stand in direct conflict with the need to generate concise information about a fault scenario.

As basic inspiration and case study in Part II, we have used a highly configurable and user programmable control system with an object oriented architecture developed for industrial robots by ABB Robotics. The main characteristics of this system is closer described in Section 11.3.

We propose a fault handling scheme as an extra layer between the operator and the core control system, performing post-processing of the fault information from the system to achieve clear and concise fault information to the operator, without violating encapsulation and modularity. The post-processing consists basically of finding a cause-effect relation between the generated error messages in a fault scenario, and then to choose the most significant error message(s) according to this relation. The scheme is illustrated from the operators point of view in Figure 10.2.

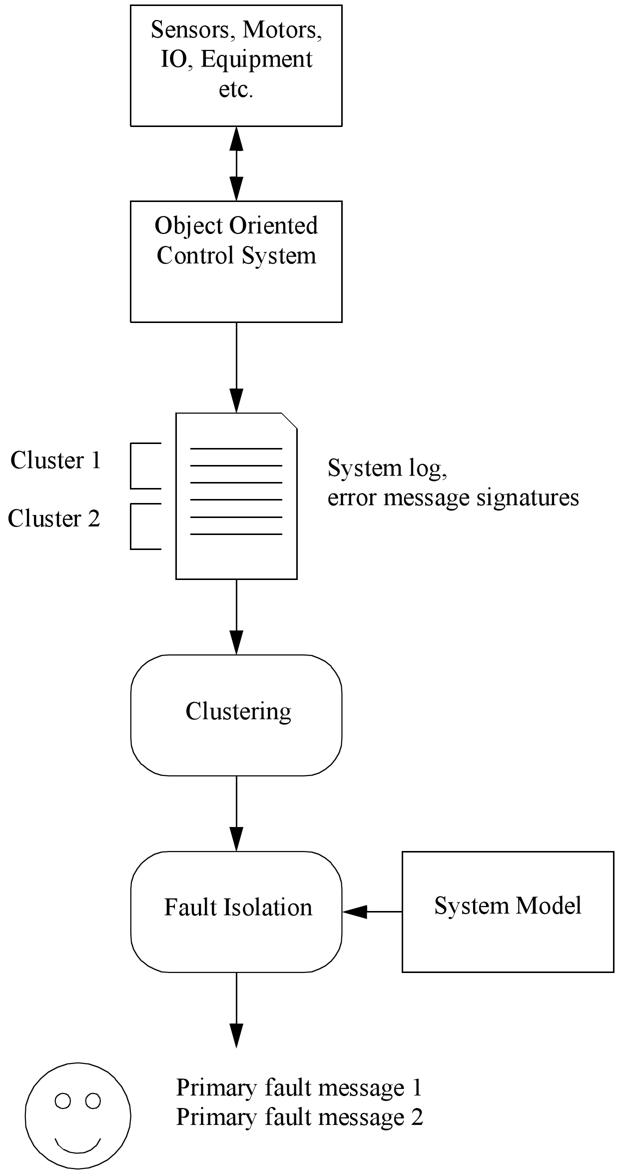


Figure 10.2: Main flow diagram of the fault isolation procedure

A prototype implementation of the approach has been made and tested on the ABB Robotics industrial robot control system. The implementation has two parts:

- A generic fault isolation tool, going under the work-name *DrRobot*.
- A system model and error message classifications specific for the ABB Robotics industrial robot control system.

An example from the ABB Robotics industrial robot control system is demonstrated in Section 10.3 as a preview of the fault isolation scheme, and the implementation and practical results are more thoroughly presented in Chapter 15.

10.1 Problem description

Our concern here is how a large-scale, configurable and safety critical object oriented control system isolates run-time faults and alarms, and specifically the issues that arise due to the object oriented structure and complexity of the control system itself. These problems were also discussed in [46].

There are two main types of run-time faults that occur: hardware faults and real-time faults. The real-time faults can be due to several reasons to be discussed in Chapter 13, and are often triggered by hardware faults themselves. With the term *fault*, we mean a run-time change or event, often in hardware, that eventually causes the system[1] to abort normal operation. The system then usually needs the attention of a human operator to resume operation. The terms fault and failure will be used synonymously in this thesis.

When a fault occurs during normal operation, the system often generates a large number of error messages due to fault propagation in the object oriented software. Error messages are sent by individual objects when the object has detected an error condition. The individual object does not in general know how close it is to the real fault or if sufficient reporting is already taking place, and hence whether it should report to the operator or not. For closely collaborating objects it is possible to suppress error messages by some kind of information passing, but this is not always feasible – it is an explicit aim of object oriented modeling to encapsulate knowledge about the internal state of objects and to achieve independence between groups of collaborating objects (i.e., encapsulation and modularity).

Since the error messages stemming from a certain fault often reflect the control system design and architecture, it can be very difficult for the operator to understand which error message that is most relevant and closest to the real fault.

Moreover, the control system that we consider here is safety critical. In case of a serious fault, the first priority is to take the system to a safe state. Only then is it possible to start analyzing what may have caused the fault. The immediate fault handling and safety aspects are not further treated in this thesis.

Our primary concern, and the starting point of this work, is a situation where we have an operational system which is running without direct supervision. Operators or service personnel called in when the system halts due to a failure are

[1] With *system* we will mean the control system, if not otherwise clear from context.

fairly unexperienced with the system and have no insight into the internal design of the control system. The basic philosophy is that improvement of the handling and reporting of faults, even simple ones, can be of great assistance; it helps unexperienced operators and unloads experienced.

Exception handling

Exception handling mechanisms are intended to help improve error handling in software, to make programs more reliable and robust. They are language constructs that facilitate error handling outside of the normal program flow and at the appropriate level. The exception constructs might also support the programmer in providing more information to the error handler code than avaliable through the normal object interface, to facilitate error recovery.

Exception handling mechanisms are to their nature low level constructs, and as such address the fault handling problem bottom up, while the scheme we propose in this thesis takes a more abstract view, and addresses the problem, mainly fault propagation, from above. The methods described in this thesis can and should be used in conjunction with low level error handling in some form. It can, e.g., be a disciplined use of return codes or extensive use of full-fledged exception handling mechanisms.

It is interesting to note that, as pointed out in [50], the goals of exception handling often stand in direct conflict with the goals of an object oriented architecture, the very same goals of encapsulation and modularity that cause the fault propagation problem addressed in this work.

10.2 Fault isolation method

Experience from ABB Robotics show that it is often possible for a skilled system developer familiar with the internal design of the system to very quickly determine the root cause of a fault by studying the logged error messages from a fault scenario. This is of course a not very surprising observation, but it is valid also for fault scenarios that causes fault isolation problems for end users. Part II of this thesis can be described as an effort to capture the necessary knowledge of the expert to perform as much fault isolation as possible automatically.

Given a system with the fault propagation problem described in Section 10.1, then one method to use for fault isolation is of course to compile a database, or to use an expert system [72, 76], linking patterns in the error message log with certain faults. A database/expert system has the disadvantage of being hard to create and maintain, since they are in most cases not a natural part of the software process, and is something that has to be done in addition to normal maintenance and development. For a highly configurable system, every installation might need a new or modified database. Also, when changes are made in the system itself, it can render an extensive database useless. We will in this thesis not consider these associative model structures further, but focus on the use of so called explanatory, or deep models (see Section 1.1).

The fault propagation in a software system as described in Section 10.1 cannot be characterized with physical entities as flow, pressure, temperature etc. exchanged between components via physical connections as is the case in the literature about fault propagation in the context of analytical redundancy model based diagnosis, see, e.g., [4, 5].

The behavior of software components does not follow any physical laws, which makes it very hard to create general component models and define component model connections that capture the behavior. It is possible to model the high-level behavior of a software system using e.g., Statecharts [33] (see also Section 11.2.4), or formulate a specialized model structure capturing normal and faulty behavior of objects and interaction of objects, and use methods similar to the ones described in Part I for fault isolation. The methods we were concerned with in Part I can also be characterized as behavioral, in that they describe the dynamic- and input/output behavior of a system.

To build such relatively detailed models of the behavior of all software components involved in the propagation of a fault would be a large and difficult task, and such models do normally not exist as a natural part of the software process, at least not in todays practice. The objections against these kind of models are hence largely the same as against a database/expert system discussed above, they are in general hard to build and maintain. We are also of the opinion that much can be accomplished with simpler means, before turning to more complex methods and detailed models.

A primary concern in Part II is to develop a model based fault isolation method that is a natural part of the software process, and hence that the model and other system specific information used in the fault isolation are easy to build and maintain in the sense that designers and developers understand it and use it for other purposes also. We propose a structural scheme for fault isolation, where error messages are explained locally, using the information available for an object while maintaining encapsulation and modularity. A guiding motto for this work has been "Try simple things first", and the use of more elaborate, behavioral models with the good property of being close to the software process preserved will be a matter of future research.

The information used for fault isolation is in two parts:

- Local, structural information put in the error messages, called the *error message signature*.

- A structural *system model*.

Together with a conceptual *explanation model*, the error message signatures make it possible to infer a probable cause-effect relation between the error messages from a specific fault scenario. The "maximal" error message(s) according to the cause-effect relation are then assumed to be most significant and can be presented for the operator. Where the local error message signature information is not enough to get a complete picture, we use the structural system model to "fill in the gaps".

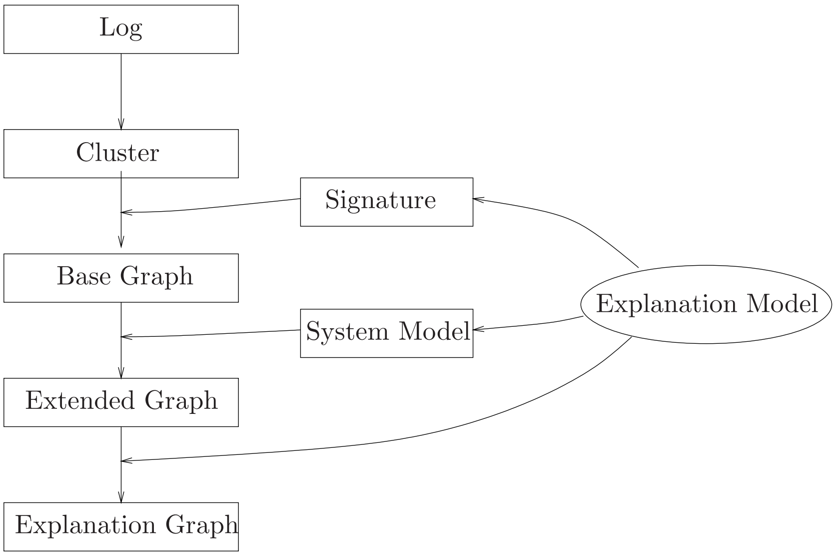


Figure 10.3: Overview of the fault isolation scheme

The main parts of the proposed fault isolation scheme are visualized in Figure 10.3. The different parts will be discussed and explained in detail in the sequel, and a brief preview of the whole procedure is given as an example in Section 10.3. To the author's knowledge, this is a novel approach. The number of error messages in a fault scenario need not be particularly large to cause problems for an unexperienced operator, the number typically ranges from 3 to 20 in the ABB Robotics application. The strength of the proposed approach does not lie in the amount of handled error messages in each fault scenario, but in the wide range of potential fault scenarios handled by a general method.

To have a system model that is close to the software process, we have chosen to use the Unified Modeling Language (UML), which in later years has established itself as a de facto standard modeling and development tool for object oriented systems. The system model used for fault isolation would then be an integral part of the system documentation (see Section 11.2 and, e.g., [55, 22]). The error messages are divided into a few types recognized and used by the fault isolation procedure, and the detailed fault information to the operator is still the responsibility of the core system, in the form of informative error messages. The fault isolation scheme for a specific system is easy, or rather natural, to maintain and to extend when the system changes, since it is an integral part of the software process and the software itself.

The part of the UML we use for the system model only provides a very rough, high-level, model of the fault propagation. More specifically, we use the class diagrams and task diagrams defined in the UML, see Section 11.2. Basically, only the dependence between (software) components is represented, and the type of dependence and type of error conditions a component experiences are not included (see also Chapter 12).

Error message signature A key observation about run-time error messages in object oriented architectures is that most of them in a natural way can be classified into two main categories. An object (class) is designed to send an error message when it encounters an error condition, and this error condition is either that a fault or discrepancy is discovered in the part of the system the object has direct responsibility for, or that another object has not performed a requested service as it should. A further refinement is achieved by noting that the non-performing object may not be known to the complaining object. This observation leads to the classification of error messages into three types:

- Internal error messages
- Relational error messages
  - Known complainee
  - Unknown complainee.

If the OO-system is regarded as a collection of collaborating, fairly intelligent, but narrow-minded individuals, these three types can be characterized with the statements "I did it", "he did it" respectively "I didn't do it".

Even though it might seem tempting, just picking the internal error message in a fault scenario is not a very good solution. One reason is that it places very hard requirements on the designers and developers of the system to define internal error messages for everything that might happen. Also, when the system is further developed and new functionality, modules or hardware is added, large parts of this analysis probably would be invalidated. By including also the relational error messages in the fault isolation scheme, a more general and flexible scheme that has better maintainability properties and is more robust against unforeseen faults is obtained. This will be further discussed in Chapter 13, where also real-time faults will be included in the scheme.

Error message design A guiding design principle for fault handling in a control system, should be that faults are reported as close to the source as possible, where the most relevant information is avaliable. This is also a requirement for the maximal error message(s) in the cause-effect relation mentioned above to be the most significant and suitable to present to a user. The burden of constructing informative and orthogonal error messages hence still lies on the designers and implementors of the system.

Another guiding design principle is that breaking of encapsulation and modularity to stop propagating error messages should be used very restrictively and only for already closely cooperating objects. Instead the fault isolation layer should pick out the most relevant messages for a specific fault scenario. However, it is still important that individual objects and collections of closely collaborating objects do not send unnecessary error messages, since the suggested approach regards the object or even the collection[2] as an atomic unit and have very limited mechanisms to determine what messages that are most relevant from such a unit. The designers and implementors of the system hence still needs to make an effort to inhibit the system from sending unnecessary error messages when possible.

10.3 Example

To give a preview of Part II, we will briefly go through an example where the implemented fault isolation scheme is applied to a real fault scenario from the ABB Robotics industrial robot control system. We will follow the scheme outlined in Figure 10.3.

In the scenario, the robot is running a user-defined program, that involves IO-communication with external equipment via the built in CAN-bus. The bus suffers a fault, that gets reported to the message log at time `13:40.52`. Some time later the program needs to access the bus, which leads to a propagation of error conditions, causing several error messages from different parts of the system and finally an emergency stop. The relevant part of the message log is shown in Figure 10.4. The message log contains in general many messages that have no connection with the specific fault scenario at hand, and the relevant part of the log must be picked out. This is called *clustering* of the log, cf. Figure 10.3.

The set of error messages can, even after clustering, be hard to interpret without insight into the internals of the control system. In this case the most relevant message, informing us that the bus has failed, is the first in the cluster, but that is not necessarily true in general (see Chapter 13). It is also not self-evident that the rest of the messages do not correspond to a real fault, but are only the result of fault propagation in the control system software.

As explained in Section 10.2, an error message is sent by an object and can be interpreted either as a complaint on another object, or a statement that something is wrong with the object itself. We call this information for the error message signature (cf. Figure 10.3), and it can be used to form a graph of how the fault has propagated in the system. This graph is called the *(initial) base graph* for the fault scenario and has error messages as edges and complainer and complainee objects as nodes. The base graph corresponding to the message log in Figure 10.4 is shown in Figure 10.5(a). The edges between two nodes denotes relational error messages and should be read "complains on". The one arrow to self adorned with *int* in the figure is used to denote an internal error message (Section 10.2).

The arrow with the diamond shaped head between two of the nodes in Figure 10.5(a) is the UML notation for generalization (see Chapter 11), and basically means that the two nodes in the base graph should be interpreted as the same instantiated object.

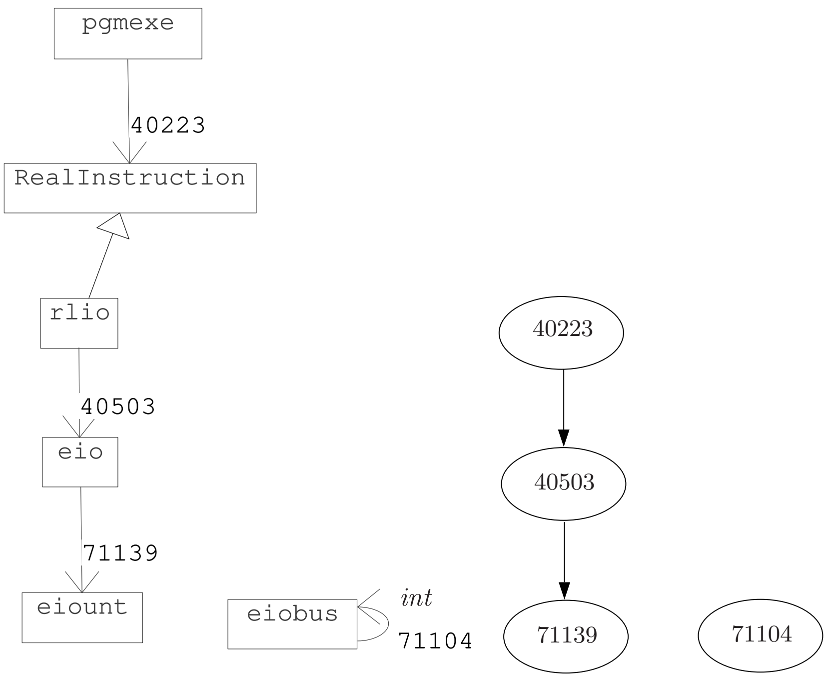
[2] In UML-terms this will be a *package*.

```
All log messages
Date of saving: 19981120 15:42

1.  71104 Error on I/O Bus 0342 13:40.52
Description\Reason:
- An abnormal rate of errors on the
  BASE Bus has been detected.

2.  71139 Access error from IO 0342 13:41.7
Description\Reason:
- Cannot Read or Write signal s1 due to
  communication down.

3.  40503 Reference error 0342 13:41.7
Device descriptor is
not valid for a digital write operation

4.  40223 Execution error 0342 13:41.7
Task MAIN: Fatal runtime error

5.  10020 Execution error state 0342 13:41.7
The program execution has reached
a spontaneous error state

6.  10005 Program stopped 0342 13:41.7
The task MAIN has stopped. The reason is that
an external or internal stop has occurred.
```

Figure 10.4: The relevant part (cluster) of the message log for the fault scenario in Section 10.3.

Note that the two last of the original six messages in the cluster in Figure 10.4 are not present in the base graph in Figure 10.5(a). These messages are so-called *operational* messages, that informs the user of major state changes in the system. In this case that the program execution has reached a failure state and is aborted. These kind of information messages are sent by high-level objects with general resource management tasks in the system, and are not part of the fault propagation from object to object as described in Section 10.2. The operational messages can be used for clustering of the log, in that they mark the occurrence of a fault scenario. In the industrial robot application, the operational messages provide no detailed information of the basic fault, though, and are simply ignored once the message log cluster is established.

By looking at the base graph in Figure 10.5(a), we see how the fault has propagated from object to object as manifested in the three relational error messages. It does not seem especially far-fetched to assume that a relational error message complaining on one object, e.g., `40503`, is explained by an error message sent by that object in turn, i.e., `71139`. This assumption is basically what we call the *(object) explanation model* (cf. Figure 10.3). According to this assumption, we can from the base graph directly form a cause-effect relation (also called *explanation relation*) on the error messages, where an error message is related to the error messages that



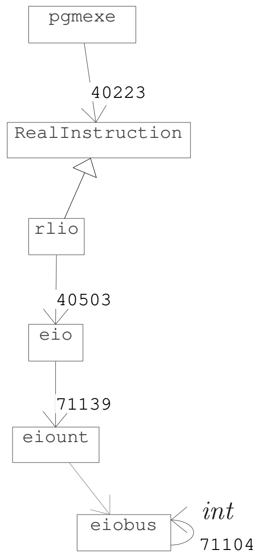
(a) Initial base graph. (b) Initial explanation graph.

Figure 10.5: The initial base graph and explanation graph for the fault scenario in Section 10.3.

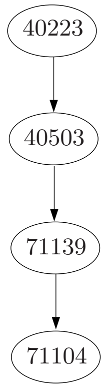
explains it. The visualization of this relation will be called the *explanation graph*, with error messages as nodes and edges that should be read "explained by". The explanation graph for the base graph in Figure 10.5(a) is shown in Figure 10.5(b).

As can be seen in Figure 10.5(b), the error message relation constructed from the initial base graph does not contain a unique maximal element in this case. This of course depends on that the initial base graph in Figure 10.5(a) is not connected; the chain of relational error messages does not have a direct connection to the internal error message stating that the bus has failed. By having a structural *system model*, that contains the information that the class `eiount` depends on services provided by the class `eiobus`, we can *extend* the initial base graph as shown in Figure 10.6(a) (cf. Figure 10.3). The added (*derived*) "complains on" edge in the base graph is shown in grey, to distinguish it from the edges corresponding to real error messages.

Now the chain of propagating error messages can be clearly seen, and the corresponding explanation relation and explanation graph in Figure 10.6(b) has a unique



(a) Extended base graph.



(b) Explanation graph.

Figure 10.6: The extended base graph and corresponding explanation graph for the fault scenario in Section 10.3.

maximal element; the one stating that the CAN-bus has failed.

10.4 Model classification

In Section 10.2, a fault isolation scheme aimed at the problem described in Section 10.1 was outlined. In this section, we make an attempt to put the outlined scheme in perspective, by discussing the information sources avaliable for a model-based automated fault isolation scheme aimed at the problem described in Section 10.1. We also propose a simple classification of the avaliable information sources. Our focus is on deep models and model-based approaches, for reasons discussed in Section 10.2.

The fault isolation scheme presented in this thesis relies on models of the control software. Such models may be complemented also with models of the physical environment that the software depends on or controls, such as processors, internal and external buses, cooling fans, power supplies, motors, external equipment controlled by the software etc.

The possible character of the software and environment models can be classified in three categories as depicted in Figure 10.7. A static structure model contains

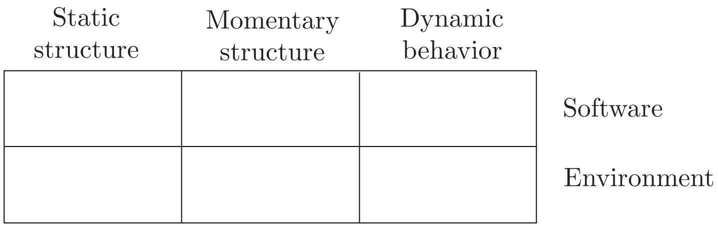


Figure 10.7: Classification of system information

knowledge of entities common to all installations and executions of the system and the relationships and dependencies between those entities. For the software, UML class diagrams (see Section 11.2.3) are used to model static structure. Similar models for hardware could be envisioned. For example, if a model states that the motor is dependent on the cooling fan, then the overheating of the motor could be inferred to the failure of the fan. Models of the environment will not be further treated in this thesis, but digraphs have been used for this purpose in the context of analytical redundancy methods, see, e.g., [12]. For the use of structure models in AI model-based diagnosis, see, e.g., [17].

The *momentary structure* of the system is a snapshot of the system at a specific time instance. The momentary structure of the software can be modeled by UML object diagrams, see Section 11.2.3. For a model of the environment, the time scale would usually be on an installation basis, but also shorter time scales could be needed, e.g., if certain equipment gets connected and disconnected in run-time. The momentary structure of the software is on the other hand usually very variable. This is discussed in detail in Chapter 13 and pretty much the rest of Part II.

Given the momentary structure of the (software) system at a certain time point, models of *dynamic behavior* would make it possible to reconstruct the momentary structure of the system at other time points. Dynamic behavior is often expressed by means of structured state machines such as statecharts. This and other possible structures for momentary and dynamic models are discussed in the context of the UML in Section 11.2.

For a specific fault scenario (see Chapter 13) the models outlined above can be constructed from four sources of information:

- an abstraction of the control software;
- a core dump;[3]
- the system configuration database;
- the error log.

[3]Strictly speaking the software may not dump core but rather suspend. However, the suspended system can be conceptualized as a core dump.

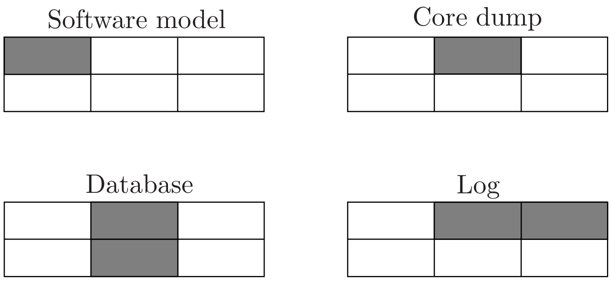


Figure 10.8: Sources of information

The current implementation only makes use of the first and last of these.

These four information sources contribute to the three kinds of models mainly as depicted in Figure 10.8. In the figure, we have assumed that the *software model* is static, since that is the case in the current implementation of the fault isolation scheme. In our case study, the static software model was (manually) reverse engineered, but developing the software model should be an integrated part of the software design.

The *core dump* provides momentary information about the software and can be abstracted into UML object diagrams. Note that the core dump does not describe the state of system at the time of the fault – because of the safety-critical nature of the application a number of actions may have to be taken before the software stops or suspends. However, the core dump provides approximate information about the state of the system at the time of the error.

The *database* contains configuration information and provides partial momentary information about both software and the environment (and the coupling between the two). For instance, the database typically contains descriptions of the physical environment of the system. It also provides information about the number of instances of classes that mirror physical entities.

The *log* provides momentary and dynamic information about the control software. Apart from error messages, the log also contains limited information about state transitions of major software entities. The log is discussed in detail in Section 13.2.1.

10.5 Limitations of the proposed approach

The proposed approach relies on information from two sources, the error message signatures and the system model. The more reliable information comes from the error message signatures, since they reflect the actual state and run-time object structure in the system at the time of the failure.

How much information that should be put in the error message signature is a design decision, that has an impact on the algorithms used for the fault isolation.

Another design decision is how "densely" the error messages should be distributed over the system. More messages make us less dependent on the system model, but it is more work involved in maintaining and defining the messages, and they will probably not be as informative for an operator. The design decisions made about the error message signatures and their implications in the implemented approach are discussed in detail in Chapter 13 and subsequent chapters.

For the system model in the form of UML class diagrams to be useful for fault isolation, the system and system model need to fulfill that the static class structure reflects the run-time object structure well. To this end, two properties are important:

- Many classes in control systems are highly specialized, and have few instantiations in the run-time system.
- Too general superclasses in the system model are avoided.

Why too general superclasses need to be avoided in the system model will be made clear in the sequel.

Even if the complete run-time object structure often is very dynamic and constantly changing, there is mostly some "major players" among the objects that are always present. If these objects have the main responsibility for error reporting, they can provide enough similarity with the static class structure for the system model based on class diagrams to be useful for fault isolation. A more static run-time structure can also be found by abstracting the system model above the level of classes and use the package level for explaining error messages, as further described in the sequel.

Certain design patterns, like the recursive Composite pattern (see [25]) causes the run-time object structure to be dramatically different from the static class structure. Such patterns are rare in today's practice, but as the use of patterns get more common, the issue needs to be addressed in the near future. This will be a matter of future research.

The more we need to rely on the system model, the more sensitive the approach is to the quality of the model and the systems fulfillment of these properties. In our experience with the ABB Robotics application, the required properties are often fulfilled, which also have inspired this work in the first place. The required properties of the system and system model are further discussed in Chapter 13 and subsequent chapters.

There is a wide class of faults that are not dealt with in the suggested approach, and has to be handled otherwise. For example, during the setting-up and running-in of a new installation, the system configuration is not yet fixed and fault-free, and so called configuration faults due to impossible and incomplete configuration variables are then very common. During this phase, experts on the system are present and hence these faults are not considered further in this thesis.

Faults in the design and software bugs are also not addressed here. Dependability of software systems and architectures for fault tolerant systems, tolerant against both hardware and software faults, is treated by Laprie et al. among others, see e.g. [42, 43].

10.6 Outline of Part II

In Chapter 11, we introduce the basic concepts of object orientation and the Unified Modeling Language (UML). The ABB Robotics industrial robot control system is also briefly described. The very short Chapter 12 formalizes the system model and specifies what parts of the UML that are used in the fault isolation scheme. Chapter 13 discusses the assumptions underlying the fault isolation scheme, the classification of error messages and the formal representation of a fault scenario in the form of a base graph. The use of the system model to extend the base graph, and how the base graph is used to form a cause-effect relation between the error messages is treated in Chapter 14. The implementation and several examples from the ABB Robotics control system are demonstrated in Chapter 15.

Chapter 11

Background

11.1 Object Orientation – Brief Motivation

This section gives a brief motivation for and introduction to object oriented (OO) methods in software development. For a thorough treatment, see e.g., [66].

The traditional structured software development methods (see, e.g., [21]) have their main focus on the data-flow and the desired functions in an application and work with functional decomposition. Object oriented software development methods on the other hand put the focus on the main artefacts, objects, of the problem domain under consideration. An *object* is a collection of data and the operations that manipulate and manage that data. The data and the manipulations associated with a real world object are kept together in the object oriented abstractions, which makes them more intuitive and more closely coupled to the problem domain. This has several advantages, some of which are:

- The coupling between analysis, design and implementation is better and made explicit.
- Complexity is better handled.
- The maintenance properties are improved.

The better coupling between analysis, design and implementation is achieved since an object in all these phases is conceptually the same, and also has a direct coupling to the problem domain.

An important design goal is that knowledge concerning a certain aspect or logical part of the system, i.e., data and the operations to manipulate it with, should be collected in one place, i.e., encapsulated in an object or collection of objects. A related design goal is that the software should consist of modular parts, that are as independent of each other as possible and communicate only via well defined interfaces. The clients of such a module should know only the interface and not how the provided services are performed and implemented. We will in the sequel use *module* as a reference to a collection of closely collaborating objects.

The looser coupling between modules and the improved abstraction facilities compared to structured methods makes the OO-method scale better to larger and more complex systems. With functional decomposition methods, it is common to have one or a few very complex master subroutines that know everything about everybody. It simply is easier to design several half-smart objects, that are aware of only their limited world, than one really smart object that knows everything.

Since the software in OO development is constructed around the main artefacts of the problem domain, the software structure is also more robust against changes in requirements and extensions of functionality. Practically every user of structured methods and functional decomposition has experienced that a small change in requirements or an added function has forced a major restructuring of the software architecture.

An example of an object could be a sensor, with the data, or attributes, measurement range, update times, location and current value. Operations could include retrieval of value and perform self-check. Many objects in a system will be similar, a system might for example have many sensors sharing the same attributes and behavior. Objects are hence organized in abstract data types called *classes*. The class encompasses both the data types and the methods of the object. An object is then an *instance* of a class.

A software system often has several concurrently executing *threads of control*, where a thread can be defined as a set of actions that execute sequentially. Many objects may participate in the execution of a thread. Threads is an operating system concept, see, e.g., [11] for a thorough treatment. In the context of operating systems, some authors use the terms thread, task and process for slightly different purposes, but we will treat them as synonyms and use the term *thread* consistently.

Objects collaborate by passing messages between them and the interface for an object basically is the messages it can accept. A message consists of data and/or control information passed from one object to another. Messages are only passed between objects whose classes have an association. How this association is represented in an UML model of the system is shown in Section 11.2.3. An instance of an association is called a *link*, i.e., the relationship association-link is analogous to the class-object relationship.

The purpose of the message can be a request for a certain service or action, or to distribute data. There are several possible implementations, e.g., an ordinary function call, where the called object then uses the same thread of control as the caller, or different schemes of communication between concurrent threads. Communication between concurrent threads is further discussed in Section 11.2.1.

Inheritance and associations will be further discussed in the framework of the Unified Modeling Language in Section 11.2.

11.2 The Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a standard notation for object-oriented systems developed by the Object Management Group (OMG) [56]. The OMG

- Use case diagram
- Class diagram
- Behavior diagrams
  - Statechart diagram
  - Activity diagram
  - Interaction diagrams
- Implementation diagrams
  - Component diagram
  - Deployment diagram

Table 11.1: Diagrams defined by the UML

is a non-profit association consisting of commercial companies, universities and others, with the task to produce vendor independent standards and specifications for object-oriented software.
In the UML standard documentation [55], the OMG states the following:

> The Unified Modeling Language is a language for specifying, visualizing, constructing and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.

The UML has in later years more or less established itself as the new de facto industrial standard notation for object oriented modeling and development. Computer tools for visual modeling with the UML is also emerging, and the most well known is Rational Rose™ [64]. Rational Rose™ is a tool for modeling, design and documentation of object oriented systems, and can be a well-integrated part of the software process. The tool is also capable of, e.g., code generation. Rational Rose™ is used to create and hold the system model in the current implementation of the fault isolation scheme. See further Section 15.1.

In this section we give a short introduction to the UML, with a focus on the parts that we will need for the system model used in fault isolation. For a complete description of the UML, see the standard documentation itself [55], or a more accessible book on the subject, e.g., [22]. The subset of the UML used in the fault isolation approach is formally specified in Chapter 12.

For representation and visualization of a system model, the UML itself employs a model/view approach, where an underlying consistent model is upheld and several different views of that model are avaliable to the user. The avaliable views of a system model can be divided in four main aspects, that perhaps are best explained by the graphical diagrams defined by the UML, as listed in Table 11.1.

The UML defines *stereotypes* as an extension mechanism, that allows the user to provide extensions and nuances to the model not present in the base UML. Almost all model elements in the UML can be stereotyped. The stereotyped elements can also be given a graphical representation by user-defined icons instead of the predefined icons. We will use stereotypes in this work to model certain asynchronous communication schemes that is not explicitly supported by the UML.

The class diagrams, together with the statechart diagrams, are often referred to as the logical model of the system, whereas the implementation diagrams constitute the "physical" model, in the sense that they describe the organization and deployment of the implemented system. The rest of the diagrams are used to describe the behavior of the system, either in the large (use case diagrams) or as samples of more detailed behavior (interaction and activity diagrams).

In Section 11.2.1 we discuss some general issues with concurrently executing threads. The different UML diagrams in Table 11.1 are discussed in Section 11.2.2 to Section 11.2.5.

11.2.1 Concurrency

A software system often has several concurrently executing threads that communicate by some sort of synchronous or asynchronous message passing. In the UML, a task is rooted in, or owned by, a so called *active object*, as is described in Section 11.2.3 about class diagrams. We also need so called *task diagrams*, that explicitly describe the run-time collaboration of threads (see, e.g., [22]). This will be discussed and demonstrated in Section 11.2.5.

Communication between threads can take on many forms, both regarding the scheme on a conceptual, logical level and regarding implementation. The possible implementations include asynchronous mail via the operating system, a synchronization of several threads (a rendezvous), shared memory areas and Remote Procedure Calls (RPC) in a distributed system, see e.g., [11].

The implementation issues are not of particular interest to us, and we will assume that an inter-process communication system (IPC) supporting asynchronous message passing is used in the system. Messages sent by one thread to another are buffered in a queue and explicitly retrieved by the addressee thread. The IPC can be supplied by the operating system or otherwise.

A usual construction is that a specific object *owns* a thread of control and serves as an interface for the thread to the rest of the system. The object retrieves IPC messages from a message queue, decides how the message should be handled and dispatches it via an ordinary function call. The callee hence only knows the thread it communicates with, and not what particular objects that handle its request.

We will now describe two asynchronous communication patterns of special interest to us in the sequel, namely *subscription* and *enqueueing*.

- In the *Subscription pattern*, a client thread subscribes to events in another thread. Examples of events may be that a certain button is pressed, an IO-signal reaches a specific value, a timeout is triggered or a computation is finished. When the thread supplying the service detects the desired event, it distributes a message, also called a notification, via the IPC to all subscribing threads.

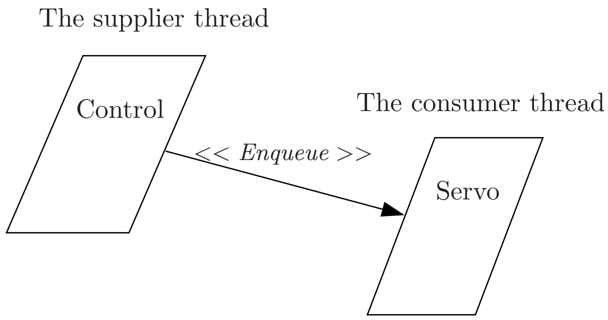


Figure 11.1: Modeling the Enqueue-pattern with UML task diagrams using the stereotype << *Enqueue* >>.

- When a thread needs a stream of data to perform its task, and that data is supplied by another thread, then the *Enqueue pattern* can be used. A supplying thread provides the consuming thread with a stream of data in the form of messages. The consuming thread does not know about the supplying thread, and only expects its message queue to contain the data it needs when it needs it.

We model these patterns in the UML using stereotypes, see Figure 11.1 for a UML task diagram modeling the Enqueue-pattern. The situation is that the thread Control supplies the thread Servo with a stream of data, e.g., set-point values for a mechanical servo to make a robot arm follow a trajectory. The fault propagation that these patterns can give rise to is discussed in detail in Chapter 13. See further Section 11.2.3 and Section 11.2.5 for modeling of thread collaborations.

11.2.2 Use case diagram

A use case diagram consists of use cases and actors and is typically used to specify or characterize the functionality and behavior of the whole system interacting with one or more external actors. Actors are the users and other systems that may interact with the system. Use cases are mainly developed in the early stages of the system development where they help delimit the system and give a clearer picture of what it is supposed to do. See further, e.g., [40]. Use cases will not be used in this work, though, since they are quite informal and more suitable for intuitive understanding and for communication between developers, domain experts and non-technicians.

11.2.3 Class diagram

Class diagrams in the UML are a melding of the class diagrams of OMT, Booch, and most other OO methods (see [55] for a more complete story). In UML class

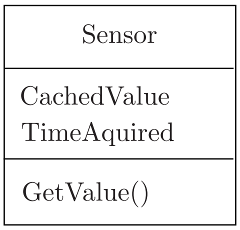


Figure 11.2: The UML notation for a class called Sensor, with attributes and methods.

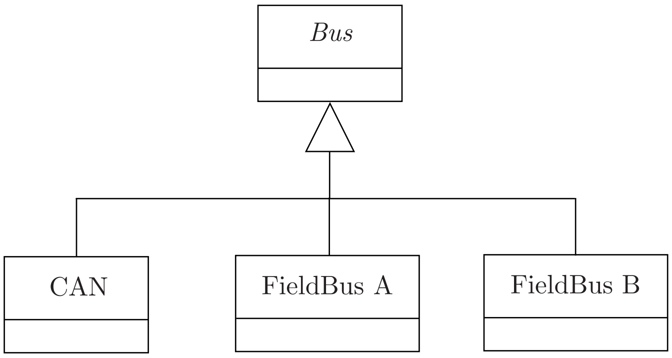


Figure 11.3: An example of the inheritance notation in the UML.

diagrams, classes (Section 11.1) are shown graphically using rectangles with the name of the class inside. Also the attributes and the methods for the class can be displayed in the rectangle, as demonstrated in Figure 11.2.

An object diagram is a graph of instantiated classes, including objects, links and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed run-time state of a system at a point in time. The use of object diagrams is fairly limited in general, mainly to show examples of run-time data structures. Object diagrams are not given an own diagram in the UML, but class diagrams can contain objects, so a class diagram with objects and no classes is an "object diagram".

A class can be a specialization of a more general class, or it could be possible to find the common features of several similar classes and create a generalization of these classes. The mechanism for this is called inheritance, or generalization, and is in the UML denoted with an arrow with a triangle head, see Figure 11.3. Generalization can be described as an "is a"-relationship. In the example in the figure, three classes are used to model three different kinds of data buses, where each of them, although different, still is a bus. The implementation of the classes might need to be different due to differences in the bus hardware and, e.g., different

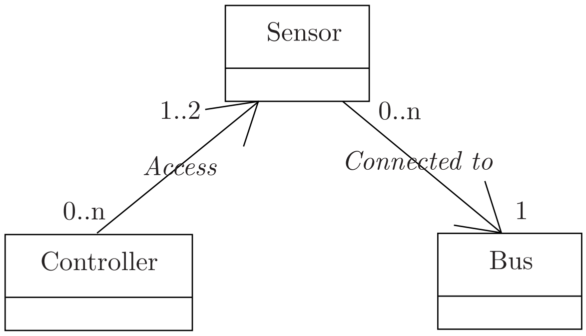


Figure 11.4: An example of the association notation in UML class diagrams.

communication protocols, but the basic functionality of a data bus is the same in all three cases. Most clients need not know what kind of bus they use, as long as the interface is the same. In the figure, the class called "Bus" collects the common features of a data bus, e.g., the interface to the rest of the system, data about current status and connected equipment, and access methods. These properties are then inherited by the subclasses. The class "Bus" is often called a superclass, or parent, and the three specialized classes are examples of subclasses, or children.

A class that only serves as a superclass, collecting certain properties to be inherited, and cannot be instantiated as an object itself is called an abstract class. This is denoted in the UML by using italics for the class name, as is also shown in Figure 11.3.

An object does not make an application, and several objects need to cooperate to solve the tasks required by the application. In UML class diagrams a client-server cooperation between objects, i.e., when an object requests some service to be performed by another object, is modeled with an *association* between two classes. The UML notation for association is a directed arrow between the classes, as is shown in Figure 11.4. The association is a "uses"-relationship, and the association hence is directed from the client-class to the server-class. In the figure, the controller object uses a sensor object to measure a certain entity, and the Sensor object in turn uses a data bus to access the physical sensor and get the values. The annotations at the ends of the associations are in the UML used to indicate the multiplicity of the instantiated objects w.r.t. each other. The 0..n at the Controller-class means that a sensor object can be used by several controller objects, and the 1..2 at the sensor end of the same association means that a controller object uses 1 or 2 sensors. It is also made clear that a sensor can only be connected to one bus, but each bus instance can serve several sensors objects.

As is illustrated in Figure 11.4, the associations can also be named to increase the understanding and intuition for the modeled system. Neither the association names nor the multiplicity labels are used in the fault isolation approach, but are mentioned here for completeness.

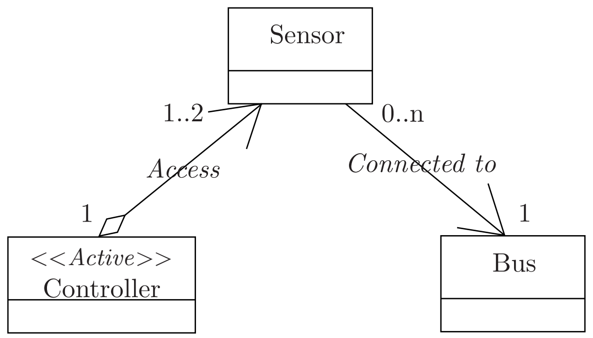


Figure 11.5: An example of the aggregation notation and modeling of active objects using stereotype << *Active* >> in UML class diagrams.

To further distinguish between different kinds of associations, the UML also defines aggregation, which is a form of association where the semantics is that the contained object is said to be a part of the aggregate. Aggregation is hence a "part of"-, or a "has a"-relationship, and is marked with a diamond on the relationship-arrow, see Figure 11.5. The difference from Figure 11.4 is that the sensors are now considered to be a part of the controller instead of a more independent entity. To further emphasize this, the multiplicity has been changed in the example so that a sensor object now belongs to exactly one controller object. This is not a requirement in the UML, though, but a question of implementation. The UML allows an object to be part of several aggregates. In the current fault isolation approach, associations and aggregations are not distinguished.

An object that owns a thread of control is in the UML called *active object*. The active object usually serves as the interface when other threads wants to communicate. The active object can, e.g., be responsible for retrieval of asynchronous messages from other threads from a message queue, and dispatching of messages to the suitable method or object. The class for an active object is depicted with the stereotype << *Active* >> as shown in Figure 11.5.

The Enqueue- and Subscribe-pattern patterns for asynchronous thread collaboration as described in Section 11.2.1 are modeled with stereotypes in UML class diagrams as is shown in Figure 11.6. In this example, the supervisor runs the planner on its own thread, and the planner in turn supplies the low-level controller with a stream of reference points. The supervisor also "listens to", e.g., the emergency button and any other safety devices.

As already mentioned, the logical model of a system can be further structured by partitioning groups of closely collaborating and related classes into modules. In the UML these collections of objects are represented with model entities called *logical packages* (see Figure 11.7). Packages themselves can have dependencies between them. The notation for a package is a folder shaped icon as in Figure 11.7, where also the notation for package dependency is shown. This type of dependency means

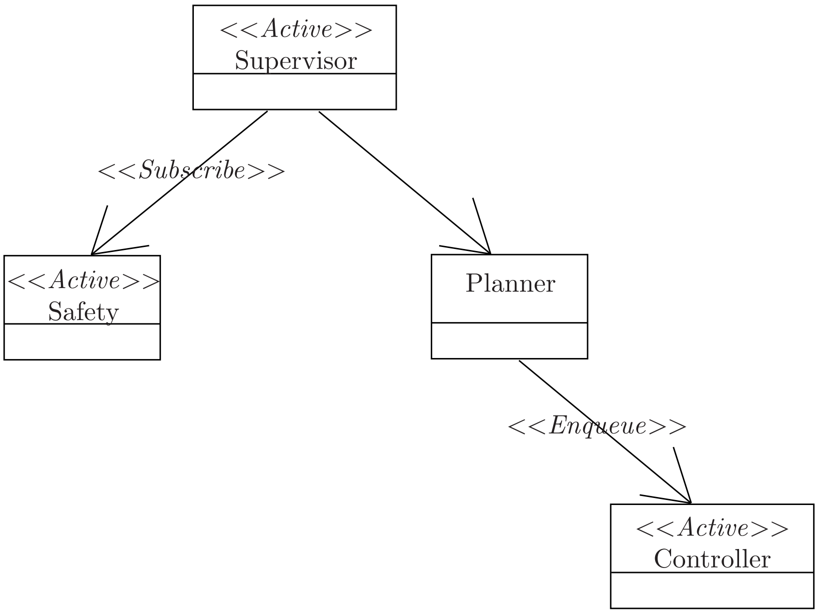


Figure 11.6: Example of the Enqueue- and Subscribe-patterns for asynchronous thread collaboration.

that the classes contained in the client package can inherit from, use, and otherwise depend on classes that are exported from the supplier package. A package models a specific subject or concern in the system, and supplies a more high-level model of the system architecture than the classes and the class relationships.

The primary contents of a package is the classes and the class relationships, but a package can also contain class diagrams, and other packages. A class always belongs to a specific package and class inheritance hierarchies are usually within a single package.

If a package contains common classes that are used by most packages in the model, the package can be declared *global*. This means that all other packages in the model can use it, without the need to explicitly specify a package dependency.

As already mentioned, the classes, class relationships, packages and package dependencies visualized by the class diagrams is referred to as the logical model of the system. The UML upholds an internal consistent model of these entities.

11.2.4 Behavior diagrams

The different behavior diagrams of the UML are used to describe the dynamic behavior and parts of the run-time object structure of the system. They are not used in the current fault isolation approach, though, and are only briefly described below.

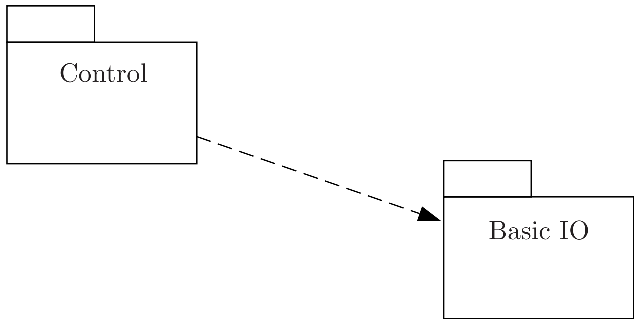


Figure 11.7: Example of packages and package dependency.

The *statechart diagram* is based on the statecharts formalism of David Harel [33] with some minor modifications. Statechart diagrams are used to specify the behavior of objects of a specific class in a basically finite state machine style. This kind of information could be useful for fault isolation, since objects often react differently to certain conditions and events depending on which state they are in. The possibility to extend the current fault isolation scheme with more dynamic information is under consideration.

The *activity diagram*, with many features in common with Statecharts, is similar to work-flow diagrams and data-flow diagrams well known from, e.g., the structured software development methods [21].

Interaction diagrams are usually used to show how multiple objects collaborate to solve a specific task. An interaction diagram can be seen as an instance of a class diagram, and may reflect the run-time structure of the software much better than the static class diagram. Hence it could be advantageous to use them for fault isolation, but in our experience so far, the main problem is that they are usually incomplete. Each interaction diagram is an instance of a class diagram and hence models a certain case of object collaboration, whereas the class diagram covers all possible collaborations.

11.2.5 Implementation diagrams

A *deployment diagram* shows processors and hardware devices and the physical connections between them. It also shows the allocation of tasks to processors. Each UML model contains a single deployment diagram.

The *component diagrams* are intended to show the physical organization of the software, including source code components, binary components and executable components and their dependencies.

We will use component diagrams to draw task diagrams, showing explicitly how the different threads in the system collaborate and depend on each other. In the UML as defined in [55], this information should be depicted in the deployment diagram. Unfortunately, the UML tool we use, Rational Rose™ [64], does not

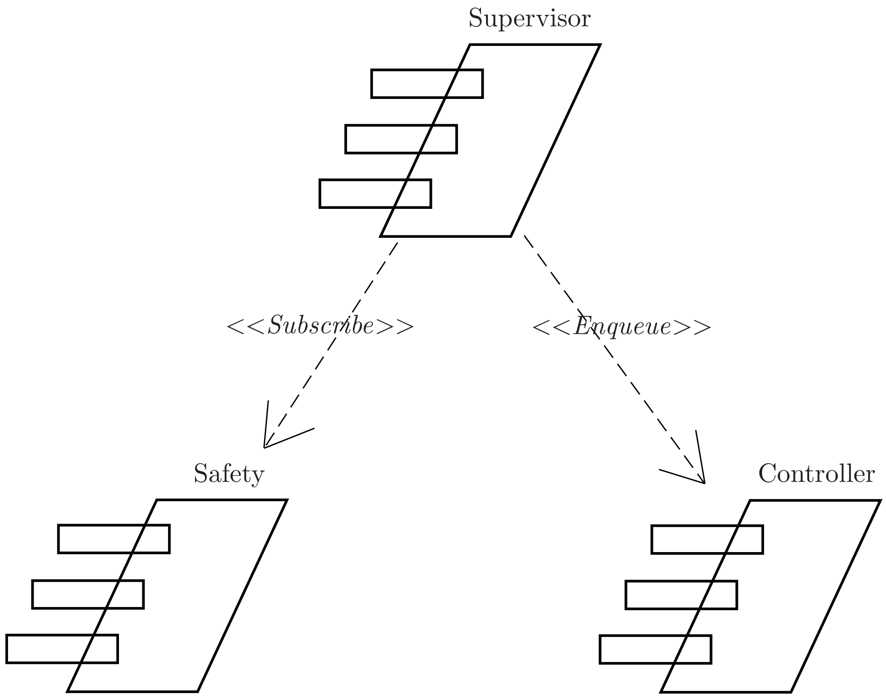


Figure 11.8: Rudimentary task diagram drawn with components.

support this part of the UML, and to avoid having a task model separate from the rest of the model, we use the component diagrams as a substitute. For our current needs, this is sufficient.

An example, that corresponds to the example in Figure 11.6, is shown in Figure 11.8. The interpretation is that the Supervisor thread is aware of both the Safety and the Controller thread, and requests services from them, but not the other way around. The Supervisor thread feeds the Controller thread data, e.g., a trajectory to follow, and subscribes to some event/events from the Safety thread, e.g., if some safety condition gets violated.

We will call the model entities modeling threads in the task diagram in Figure 11.8 for *thread-types*, as opposed to an instantiated thread in the run-time system. This corresponds to the relation class/instantiated object since a thread in the system model can have many instances in the run-time system.

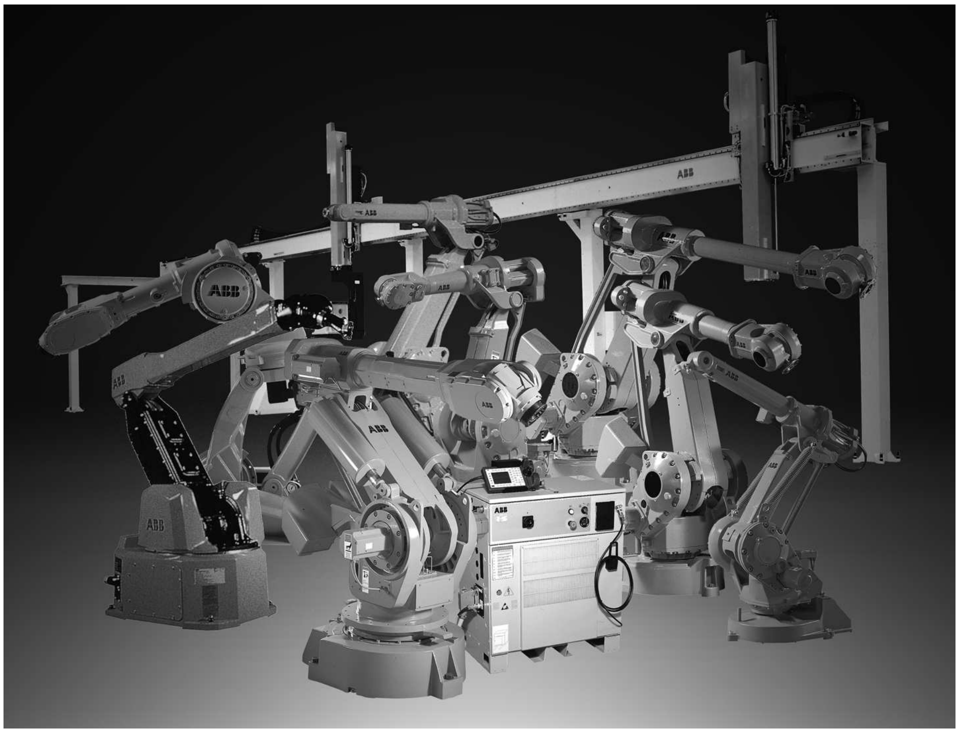


Figure 11.9: Examples of ABB Robotics industrial robots.

11.3 The ABB Robotics industrial robot control system

As basic inspiration and as a case study for this work, we have had the opportunity to work with a control system with object oriented architecture for industrial robots developed by ABB Robotics. We believe that the system has many characteristics in common with other control systems with object oriented architecture, and the fault handling scheme described later on is thus not limited to robotics.

ABB Robotics has a family of industrial robots used for, e.g., welding, painting, glueing, material handling, pick-and-place and flexible automation in a wide range of industries. The control system is designed to handle all of these robots, which means that the control system is highly configurable depending on which robot a particular system is controlling, and what extra equipment and devices that are used in the particular installation. The configuration is stored in a database for each control system. A selection of robots controlled with the same software is shown in Figure 11.9.

To be able to perform all these tasks, the system is programmable by the user, using a special programming language called RAPID. Both the robot itself and external equipment can be controlled by RAPID programs from the control system, that in turn can communicate with, e.g., PLC's and be connected to local networks.

The control system is multi-threaded, i.e., there are several concurrently executing tasks, and on several processors. The threads communicate both asynchronously and synchronously via an IPC messaging facility. The communication patterns subscription and enqueueing described in Section 11.2.1 are commonly used.

The objects in the system are both pure software objects as well as objects corresponding to hardware. Many of the hardware devices that the control system handles, have a corresponding software object that serves as the interface for that device to the rest of the system. We will call such objects (classes) on the border of the system *mirror objects (classes)*. Examples of hardware devices are power units, cooling fans, servos, external buses and IO-cards. One of the tasks for the mirror objects is to supervise their hardware and report when a problem is detected.

Chapter 12

System Model

The model structure used in the fault isolation scheme is a subset of the UML, and in this chapter we make an effort to formalize exactly what parts of the UML model, as presented in Section 11.2, that are used. Our system model consists of two separate parts, a *class model* and a *thread model*.

12.1 Class Model

The class model we use, that actually also contains package information, can be formalized as the five-tuple

$$< \mathsf{Classes}, \rightarrow\!\!\triangleright, \mathsf{A}, \mathsf{P_C}, \mathsf{PD} > .$$

$\mathsf{Classes}$ is the set of all class names in the UML model of the system. No information like attributes, methods, stereotypes etc. is used, but only the class name. The run-time system contains many instantiated objects from $\mathsf{Classes}$.

The inheritance, or generalization, relation between classes in the UML is present in our system model as the transitive and anti-symmetric relation

$$\mathsf{c} \rightarrow\!\!\triangleright \mathsf{c}' \in (\mathsf{Classes} \times \mathsf{Classes}).$$

Compare with the UML notation in Figure 12.1. The use of multiple or recursive inheritance is currently not allowed.

For use in the sequel, we will introduce some notation in connection with the inheritance relation between classes. The following three functions have the definition and value domain $\mathsf{Classes} \rightarrow 2^{\mathsf{Classes}}$, where 2^{Classes} denotes the set of all subsets of $\mathsf{Classes}$.

$$\begin{aligned}
\mathrm{desc}(\mathsf{c}) &= \{\, \mathsf{c}' \in \mathsf{Classes} \mid \mathsf{c}' \rightarrow\!\!\triangleright \mathsf{c} \,\} \cup \{\, \mathsf{c} \,\} \\
\mathrm{asc}(\mathsf{c}) &= \{\, \mathsf{c}' \in \mathsf{Classes} \mid \mathsf{c} \rightarrow\!\!\triangleright \mathsf{c}' \,\} \cup \{\, \mathsf{c} \,\} \\
\mathrm{rel}(\mathsf{c}) &= \mathrm{desc}(\mathsf{c}) \cup \mathrm{asc}(\mathsf{c})
\end{aligned}$$

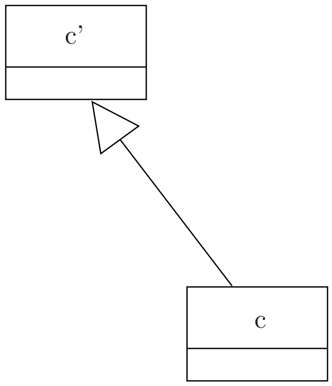


Figure 12.1: Inheritance relation in UML

The function desc(c) maps c on the set of c itself and all descendants of c, i.e., all classes that inherits from c. Correspondingly, the function asc(c) maps c on the set of c itself and all ascendants of c, i.e., all classes that c inherits from. These mappings, and their union rel(c) ("relatives" of c), are illustrated in Figure 12.2.

The reflexive and symmetric closure of the relation $\multimap$ is an equivalence relation, and by $[\mathsf{c}]$ we denote the equivalence class[1] that contains c. Note that rel(c) and $[\mathsf{c}]$ in general are different, e.g., in Figure 12.2, $[\mathsf{c}]$ consists of all five classes, including the sibling of c.

The association and aggregation relationship in the UML is represented with the *association relation* A

$$\mathsf{c} \xrightarrow{\mathsf{t}} \mathsf{s} \in \mathsf{A} \subset (\mathsf{Classes} \times \mathsf{S_A} \times \mathsf{Classes}).$$

$\mathsf{S_A}$ is a (finite) set of stereotypes of special meaning to the fault isolation scheme, where also the empty stereotype, i.e., no stereotype, is present. In the current implementation, the association stereotypes used are <<*Enqueue*>> and <<*Subscribe*>> (see Section 11.2.3). If two classes collaborate according to one of these patterns, but also have other, non-stereotyped, communication, e.g., via ordinary function calls, this must be modeled with different associations. In Chapter 14, we show how this is used in the fault isolation.

For later use, we introduce selector functions for the association relation. Let $\mathsf{a} = \mathsf{c} \xrightarrow{\mathsf{t}} \mathsf{s} \in \mathsf{A}$, then

$$\begin{array}{lcl} \text{client}(\mathsf{a}) & = & \mathsf{c} \\ \text{server}(\mathsf{a}) & = & \mathsf{s} \\ \text{stereotype}(\mathsf{a}) & = & \mathsf{t}. \end{array}$$

[1] Unfortunate terminology conflict between graph theory and OO. $[\mathsf{c}] \subseteq \mathsf{Classes}$.

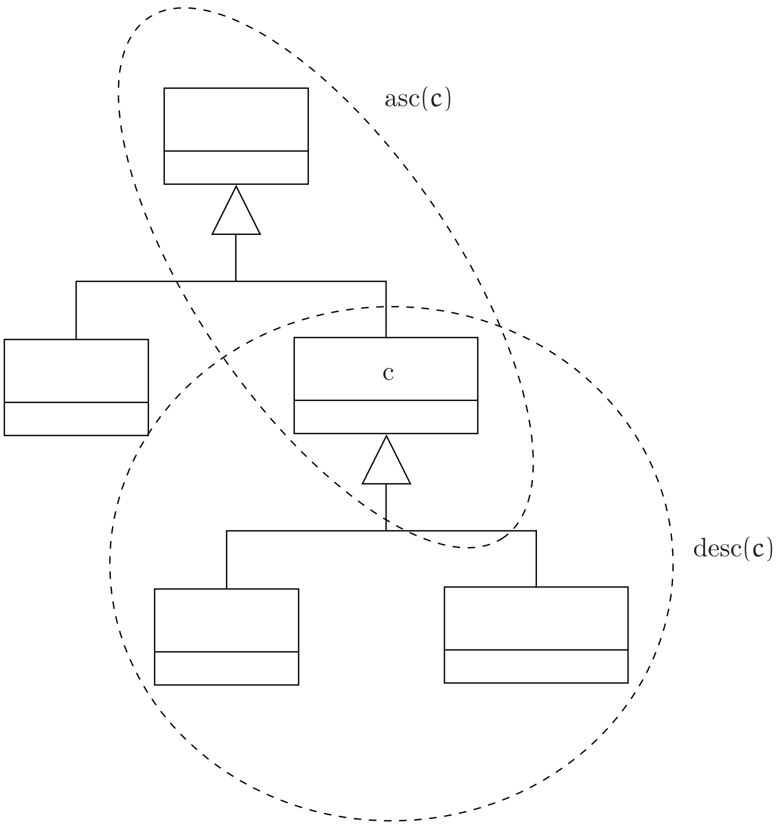


Figure 12.2: The inheritance relation and the functions desc(c) and asc(c).

The logical partioning of classes into Packages in a UML model is represented with the the *package partition* $\mathsf{P_C} \subset 2^{\mathsf{Classes}}$, where all elements in $\mathsf{P_C}$ are disjunct. The inheritance relation $\multimap$ and the package partition $\mathsf{P_C}$ are restricted so that each equivalence class $[\mathsf{c}] \subseteq \mathsf{Classes}$ is part of exactly one package, i.e., $[\mathsf{c}] \subseteq \mathsf{p} \in \mathsf{P_C}$.

The package dependencies in the UML model will also be used for the fault isolation, and we hence define the *package dependency relation* PD

$$\mathsf{c} \dashrightarrow \mathsf{s} \in \mathsf{PD} \subset (\mathsf{P_C} \times \mathsf{P_C}).$$

For later use, we introduce the corresponding selector functions. Let $\mathsf{d} = \mathsf{c} \dashrightarrow \mathsf{s} \in \mathsf{PD}$, then

$$\begin{aligned} \text{client}(\mathsf{d}) &= \mathsf{c} \\ \text{server}(\mathsf{d}) &= \mathsf{s}. \end{aligned}$$

Note that in a true object-oriented spirit, we use polymorphic selector functions, i.e., the same function names are used as for the association relation.

A class package can be defined as *global*. All other packages in the model are then assumed to have a package dependency to that package, even though the dependency is not explicitly present in PD.

12.2 Thread model

The thread part of the system model is simpler than the class model, and only consists of a two-tuple with thread components and stereotyped thread dependencies

$$< \mathsf{T_C}, \mathsf{TD} > .$$

$\mathsf{T_C}$ is the set of all threads modeled as components in the UML model as described in Section 11.2.5, and TD is the set of thread dependencies

$$c_t \overset{t}{\dashrightarrow} s_t \in \mathsf{TD} \subset (\mathsf{T_C} \times \mathsf{S_{TD}} \times \mathsf{T_C}).$$

$\mathsf{S_{TD}}$ is a finite set of stereotypes of special meaning to the fault isolation scheme, where also the empty stereotype, i.e., no stereotype, is present. As for the association relation, the stereotypes used in the current implementation are <<*Enqueue*>> and <<*Subscribe*>> (see Chapter 11). In Chapter 14, we show how this is used in the fault isolation.

The selector functions for a thread dependency relation $d_t = c_t \overset{t}{\dashrightarrow} s_t$ are defined as

$$\begin{array}{lcl} \text{client}(d_t) & = & c_t \\ \text{server}(d_t) & = & s_t \\ \text{stereotype}(d_t) & = & t. \end{array}$$

Chapter 13

Fault scenario

In this chapter, we discuss in detail the practical foundations of the chosen fault isolation scheme, and the formal classification of error messages they lead to. We also introduce the formal representation of a fault scenario, the so-called *base graph*, as used by the fault isolation scheme.

First we specify what we in Part II of this thesis mean with a fault in the context of control systems.

Definition 13.1 Fault
A **fault** is a run-time change or event, that eventually causes the system to abort normal operation. □

In our context, a fault often occurs in hardware, but can also be caused by real-time problems.

As already mentioned in Chapter 10, the occurrence of a fault often gives rise to a large number of events in the system, many of which are reported to the user. We will use the term *fault scenario* in a rather broad sense, referring to what happens with the system when a certain fault occurs. An attempt to a definition is the following:

Definition 13.2 Fault scenario
A **fault scenario** consists of the events, objects, links, threads, equipment and physical connections that are involved in the origin and propagation of a specific fault. □

In this thesis, we assume that only one fault needs to be isolated at a time. This single fault assumption of course implies that every fault scenario has exactly one basic cause. The probability of several independent faults occuring at the same time is very low in practice and is hence neglected. The time-frame for a fault scenario, within which the multiple faults must occur, is in our context, the ABB robot application, mostly just a couple of seconds and very rarely exceeding 10 minutes (see further Section 13.3). Most ideas presented in this work is though still applicable in a multiple fault situation, as is briefly discussed in Section 14.3. The single fault that has occured in a fault scenario will also be called the *primary fault*.

In Chapter 10, we primarily discussed the fault propagation between objects and modules of objects. The type of real-time problems where a thread stops executing or slows down for some reason, so that other dependent threads experience problems, are also included in the fault isolation framework. The thread-specific problem can be the basic cause for the fault scenario or just a part of a fault scenario caused by a hardware fault.

The whole approach to fault isolation in Part II of this thesis is based on some basic assumptions and facts on how a run-time fault propagates in a multi-threaded object-oriented control system. These assumptions and facts are collected and given the name *explanation models* in Section 13.1. The formal classification of error messages and the definition of error message signatures according to the explanation models is presented in Section 13.2.1. The clustering of the system log into fault scenarios have not been implemented yet, but the issue is discussed in Section 13.3. The formal representation of a fault scenario, the base graph, used in the fault isolation scheme is introduced and formally defined in Section 13.4.

13.1 Explanation Models

We will in this section discuss in some detail the occurrence and manifestation of run-time faults in a multi-threaded object-oriented control system. The high-level descriptions/assumptions presented here of how the system reacts to faults and how a fault propagates in the system will be called *explanation models*. The concept of explanation model is admittedly somewhat vague, but we have found it a useful collective term when referring to the assumptions and facts underlying the fault isolation scheme presented in the sequel. In Section 14.3, we introduce rules for how the formal representation of a fault scenario (the base graph) is used to form a cause-effect relation on the error messages. These rules can be seen as a further formalization of the explanation models discussed below.

13.1.1 Fault Propagation

A running multi-threaded object-oriented application consists of many instantiated objects executing on several threads. The objects are partitioned into modules (modeled as packages in the UML) according to their class as described in Section 11.2.3. Faults (Definition 13.1) that occur in one part of the system cause deviations from the expected, normal, behavior. These deviations propagate to other, dependent parts of the system, by way of triggered *error conditions*. An error condition starting such an *error condition propagation chain*, is called a *root error condition*.

When an error condition is encountered by an object, the current method normally returns with an error code. It might also decide to continue its normal operation, e.g., an active objects event driven main loop. The returned error code in turn can be regarded as an error condition by the calling object, in the sense that it is a deviation from normal behavior. If deemed appropriate by the designers, the object registers an error message to a log. These error messages are treated in detail in Section 13.2.1.

For closely collaborating objects it is often possible to suppress error messages by some kind of information passing, but this is not always feasible – it is an explicit aim of object oriented modeling to encapsulate knowledge about the internal state of objects and to achieve independence between groups of collaborating objects (i.e., encapsulation and modularity).

Large control systems are often organized in a hierarchical fashion, as a natural consequence of the breaking down of large, complex tasks and responsibilities in subtasks and more limited areas of responsibility. "On the top", you typically have objects responsible for the overall activities and high-level functions of the system, like resource management, managing the overall state of the system, supervision and user interface. An example of a possible (and of course incomplete) functional hierarchy in a control system, is the following:

1. Resource management; Supervision and safety; User interface
2. High level control and planning of system behavior;
User influence via programs or otherwise
3. Low level planning, e.g., detailed trajectory planning;
Low-level control; IO
4. Hardware interface

When an object, in our context usually a low-level object, detects an error condition that is deemed so serious that nominal operation cannot continue, the system often has to perform what basically is a software-triggered emergency stop. An emergency stop, with the ensuing shutdown of the whole system, is orchestrated "from the top", by the objects with overall responsibility for the activity in the system.

When an object or module of closely collaborating objects receives a shutdown order "from above", they of course no longer expect the same services from other parts of the system, and the fault propagation stops. The shutdown is not immediate in the whole system, though, since the order itself must propagate through perhaps several layers and long call chains. Also, certain safety measures must generally be taken immediately, before an orderly shutdown of the different parts of the system is considered. This situation often leads to a race between the two chains of events; the fault propagating as error conditions "upwards" in the system, and the shutdown order propagating "downwards". Our focus is on the error propagation from the fault source and the operator messages and alarms that it gives rise to. The messages and alarms the “downwards” chain of events causes is quite easily handled, as will be discussed in Section 13.2.2.

An error condition is, in our context, always detected by an object executing on a thread, but the error condition can be mainly associated with either the object or the thread. Our main focus in the current approach is on objects (and modules of objects) as the base for the explanation model, described further in Section 13.1.2. The explanation model needed to include thread and concurrency concepts in the fault isolation is discussed in Section 13.1.3.

13.1.2 Object explanation model

Many instantiated objects collaborate and depend on each other as described in Section 11.1 and Section 11.2.3. In a well designed system, the objects (classes) and modules, logically partitions the system into different areas of responsibility. The basic object explanation model is simply that if an object experiences problems of some sort, then it will in turn have problems fulfilling its responsibilities towards the rest of the system, causing problems for other objects. Hence, as exemplified in Section 10.3, if the problems one object experiences can be related to problems in another object, the first is assumed explained by the second.

Adhering to this basic assumption, the error conditions encountered by an object can broadly be classified in two categories (as already mentioned in Section 10.2).

Internal A discrepancy in the data or part of the system the object encapsulates knowledge of.

Relational Another object, module or thread has deviated from nominal behavior, from the detecting object’s viewpoint.

The error conditions that can be classified as internal mostly occur at the border of the system, in what we called mirror objects in Section 11.3. A detection of a failure of some sort in the device the mirror object encapsulates knowledge of would be an internal error condition. Practical examples are, e.g., an overheated motor, a bad or broken communication link or an error reported by external equipment with own supervision capabilities. Hardware faults detected by analytical redundancy methods (see Section 1.1.2) would also often fall in the internal error condition category.

The relational error messages lie at the core of the object explanation model, since they are the visible result of the software fault propagation. An error message falls in the relational category when it is sent due to a detection of an unsatisfactory behavior of another object, module or thread. The probably most common example of a relational error condition is when an ordinary method call made by one object to another returns with an error code, and hence the requested service was not performed. The object detecting the error condition is called *complainer* and the trouble-making object is called *complainee*. The object detecting an error condition notifies the user by sending an *error message*, see Section 13.2.1.

A natural thought in the case of ordinary method calls, is to put information about the error condition in the complainee in the returned error code. This has several drawbacks associated with it, though. The interface between objects gets quite complicated and unwieldy and hence hard to manage. Not the least, such an interface would also often have to be implementation dependent, directly violating very sound fundamental object-oriented design principles.

The notion of relational error condition needs to be further refined into two cases: *known complainee* and *unknown complainee*. In the latter case, the complainer has no information on what other system entity failed to perform as expected. This can be for encapsulation and modularity reasons.

Package explanation model In the object explanation model, we also include the assumptions that error conditions can propagate between closely collaborating objects, and between groups of closely collaborating objects. In the fault isolation scheme, the groups of collaborating objects are defined by the logical packages in the UML system model. The exact implication of this assumption will be made clear in Section 14.3.

13.1.3 Thread explanation model

A dimension of the fault isolation task that cannot be overlooked is given by the existence of several concurrently executing threads, often on different processors, in a control system of any complexity. Analogous to the case of object explanation models in the previous section, the basic assumption in the thread explanation model is that if a thread experiences problems, then other threads that depend on it will also experience problems.

The object and thread explanation models represent two different views of the same system. The classes and modules (packages) represent a logical partition of a system[1], whereas a single thread often involves objects from many different parts of the system. The thread explanation model is hence, loosely put, more action oriented than the object explanation model, and the two views can even be said to be perpendicular, in the sense that a thread involves many objects and an object may execute on several threads. The coupling between the two views is twofold:

- As discussed in Section 11.2.1, an (active) object serves as the interface to a thread. If the thread experiences a problem, it might cause a propagation of error conditions between object as described in Section 13.1.2.
- The problem a thread experiences can cause error conditions in objects executing on the thread.

Examples will be given in Section 13.4 and Chapter 15.

A thread of control may slow down, get stuck or completely exit abnormally for a number of reasons, all of which can initiate error conditions for other threads and

[1] In a well designed system, that is.

their objects. Error conditions directly corresponding to real-time and interprocess communication problems will be called *IPC-error conditions*.

The concepts of internal and relational error conditions introduced in the previous section are applicable also on the IPC-error conditions. The IPC-error conditions, in our experience from the ABB application, almost always fall into the relational category, since a thread of execution with problems most likely is not able to execute code that would detect the error condition. Since the thread view of the system is so distinct from the class view, the natural complainee (and complainer) for IPC-error conditions is the thread. Detailed definitions are given in Section 13.2.1.

Some of the specific problems a thread can experience are detailed below. It should be clear that these thread specific problems can cause fault propagation between parts of the system separate in the class diagram view, and hence that the thread explanation model is needed.

A thread can be slowed down enough to cause error conditions if, e.g., the CPU is overloaded by another thread, some job performed by the thread itself takes a long time to complete, or the computational load on the thread is too heavy. A slowed down or stuck thread can, e.g., be waiting for a hardware device that has suffered a failure, or a shared resource that is held by another thread, whose execution in turn has stopped or slowed down. Thread execution problems can also be caused by faults in the design or implementation, e.g., there can exist deadlock situations or not properly tuned timeout lengths and task priorities. A thread may exit abnormally due to some sort of operating system violation, caused by, e.g., a software bug.

Examples of error conditions caused by slow threads are queue overflows, due to the slow or stuck thread emptying its message queue at a too slow rate. The queue overflow may then be an error condition for the thread sending the "overflowed" message. Another scenario is that expected messages do not arrive on time, i.e., a message queue gets empty due to a slowed down supplier thread. This is called *starvation* and can happen, e.g., with the enqueue-pattern described in Section 11.2.1.

If two threads communicate synchronously, error conditions in the form of timeouts will occur if one thread tries to initiate a communication with a slowed down thread. This can, e.g., be the case with the subscription-pattern (Section 11.2.1), when the supplier thread tries to deliver the requested notification to a non-responding or exited thread.

For relational IPC-error conditions, in addition to the known/unknown refinement mentioned above, we need also make a distinction related to how the thread collaboration is modeled in the task diagram of the system model (see Section 11.2.1 and Section 11.2.5). In the task diagram (see Figure 11.8), the dependency arrow is from the client to the server. When the communication is asynchronous, it is not necessarily the client that experiences problems with the thread it has requested services from, but it might be the other way around. One example of this situation is the error condition described above for the subscription-pattern, where the client tries to deliver a notification to a non-responding thread. Another example, also mentioned above, is for the enqueue-pattern, when the server thread unexpectedly finds its message queue empty. These error conditions are called *backward relational*, referring to the direction of the dependency in the task diagram in the system model. The complainee can still be either known or unknown.

An exception to the relational nature of IPC-error conditions would be if a so called *watchdog-pattern* (see, e.g., [22]) is employed, where some threads are explicitly supervised by a watchdog object running on a separate thread. The problems detected by the watchdog could then be categorized as internal w.r.t. the thread (see further Section 13.2.3). In the current implementation, all IPC-error messages are assumed relational, though.

The type of fault situation where a thread problem is the root cause of a system shutdown is often harder to handle from a user/operator perspective than hardware faults, since the trouble-making thread often is not in a position to give any indication on what is wrong in the form of an error message. If that is the case, the current approach can at best identify the silent, trouble-making thread, and leave the further detailed diagnosis to others. See Chapter 14.

13.2 Messages and the System Log

Events that occur in a control system often need to be reported to an operator or user of the system, for required immediate action or status information. For our purposes, we assume such messages are recorded in a system log that is avaliable also after a shutdown of the system.

Definition 13.3 System log
The *system log* is a list of events recorded by the system in chronological order in the form of *messages*.

Remark: The system log is an abstraction, and need not necessarily be centralized in a real system, since we in the current setting are doing a-posteriori analysis of a fault scenario. □

It is very important to distinguish between the messages sent to the system log, and the messages discussed in Chapter 11 for inter-object and inter-process communication. When the distinction is not clear from context, we use the term *log-message*. How the error messages associated with a specific fault scenario is picked out of the system log is discussed in Section 13.3. The set of error messages in the log corresponding to a specific fault scenario is called a *cluster*.

We have chosen to make no use of the chronologic order with which error messages arrive in the log, other than for clustering purposes. In a system with many concurrent threads, the order of the messages is not necessarily significant.

A design principle for error messages that must be followed for the fault isolation scheme in this thesis to be useful, is that faults should be reported as close to the source as possible, where also the best information on the occured event is available.

Specifically, internal faults need to be as informative as possible, since they often are to be presented to the user as a result of the fault isolation.

The log-messages we will discuss in the section are divided in three groups:

- Run-time error messages
- Operational and status messages
- Supervisory messages

The actual run-time error messages sent in the course of the "upwards" fault propagation discussed in Section 13.1.1 and the information they need to contain, are treated in detail in Section 13.2.1. Operational and supervisory messages are often sent to the log during a fault scenario and are discussed in Section 13.2.2 and Section 13.2.3 respectively. Section 13.2.4 contains a brief discussion of unobservable faults in our context.

13.2.1 Run-time error messages

The messages sent to the log during the "upwards" fault propagation discussed in Section 13.1.1 will be called *run-time error messages*. The error messages constitute the most important information we have about a certain fault scenario, since they give a picture of what happened in the system at the time of the failure (see Section 10.4).

Definition 13.4 Run-time error message
A **run-time error message** (error message for short) is a message sent by an individual object to the system log, in response to the detection of an error condition for that object. □

Recall that an error condition is a deviation from normal behavior, when an object cannot fully fulfill its responsibilities in the system as expected (Section 13.1.1). From the viewpoint of an automated fault isolation scheme, the error messages are the observable effects of a fault scenario.

In this section we will introduce a classification of error messages, pertaining to the classification of error conditions discussed in Section 13.1. We also specify the corresponding information an error message must contain for fault isolation purposes.

According to the discussion in Section 13.1, the error messages are classified into six types, shown in Table 13.1. In the introduction to Part II, Chapter 10, we only mentioned three groups, but now we also take the IPC-error messages into account. The object error messages in Table 13.1 will also be called *ordinary error messages*.

To form the formal representation of a fault scenario, the so called *base graph*, from a cluster of error messages, we need to know the type of each error message, and the corresponding information about sending object, complainee, sending thread etc. This information belonging to an error message, will be called the *error*

- Object error messages
  - Internal
  - Relational
    - Known complainee
    - Unknown complainee.
- Thread (IPC) error messages
  - Relational
    - Known complainee
    - Unknown complainee
      - Direction server
      - Direction client

Table 13.1: The six error message classifications according to Section 13.1.

message signature. For two main reasons, the sending object should attach this information to an error message:

- Maintaining the signature becomes a natural part of the software development process, as discussed in Section 10.2.

- Instance information, both for objects and threads, can be added to the signature, as will be further discussed below in this section.

In the current implementation of the fault isolation scheme for the ABB Robotics industrial robot control system, the signature is added a-posteriori using a lookup-table, which can be done under certain assumptions further discussed below in this section.

The map from the error message cluster to initial base graph using the error message signature is quite straightforward, given the exact definition of the base graph, which will be given in Section 13.4.

In Figure 13.1 we show the error message signature and the error message classification used by the current implementation of the fault isolation scheme. It is shown as a part of a possible log-message data structure design for an object oriented control system. It is of course enough if the error message data structure for the system under consideration can be mapped into this form without loss of information.

The design is shown in UML class diagram notation (see Section 11.2.3). Some of the messages sent to the system log that are of interest to cluster the log into clusters corresponding to fault scenarios (Section 13.3) are not represented in Figure 13.1. These messages are discussed in Section 13.2.2. Also recall the example in Section 10.3.

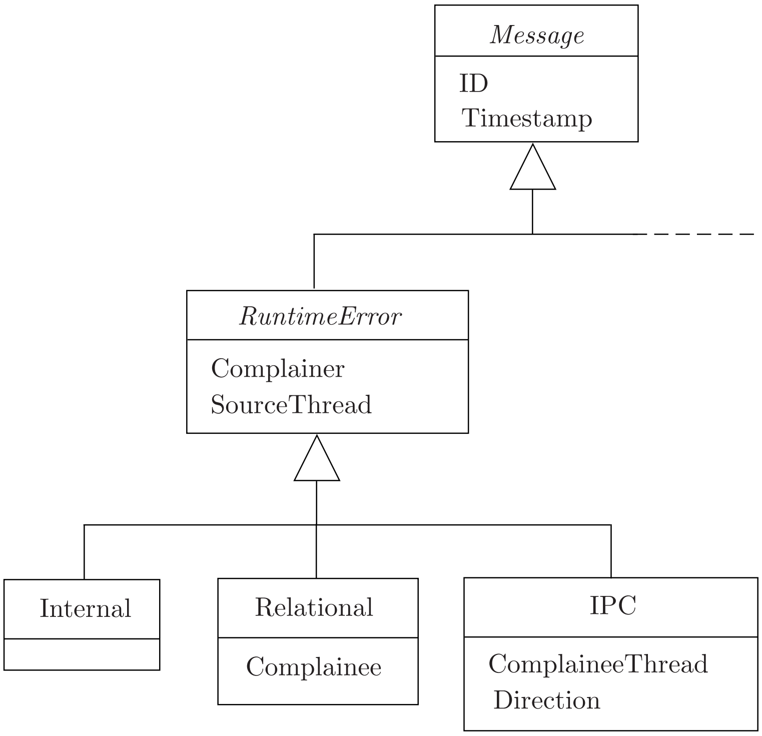


Figure 13.1: A possible error message signature design.

Class Message The top class in the hierarchy is the abstract class Message, symbolizing all messages that can be sent to the system log. The attribute ID is a unique identifier, e.g., an integer, for each log-message defined in the system. The attribute Timestamp contains the exact time a message was sent.

Class RuntimeError The run-time error messages introduced in Definition 13.4 are represented by the class RuntimeError. These error messages specify which object sent them (attribute Complainer) and also on which thread the sending object was executing (attribute SourceThread). The identifiers used must of course match with the corresponding entities in the system model in Chapter 12. The abstract RuntimeError class is specialized into three subclasses corresponding to the classification of error conditions into internal, relational and IPC-errors discussed in Section 13.1. For a relational error message (class Relational in Figure 13.1), the complainee object is identified by the the attribute Complainee. If the complainee is not known in the relational error message, the attribute is simply empty.

The object that sends an error message executes on a specific thread in the runtime system, identified by the SourceThread attribute. When representing a fault scenario formally in Section 13.4, we will consider the sending thread to correspond to the message and not the complainer object. If an error condition arise when objects on different threads communicate, that error condition should be classified as an IPC-error condition or a relational unknown.

Class IPC The class IPC in Figure 13.1 corresponds to the the IPC-error conditions discussed in Section 13.1.3. As explained there, the complainee is not an object but a thread. In the current implementation, all IPC-error messages are relational. The complainee thread is identified by the attribute ComplaineeThread, and the sending thread identified by the attribute SourceThread is interpreted as the complainer. The IPC-error message is still sent by an object, identified by the attribute Complainer. An IPC-error message can also be relational unknown, in which case the attribute ComplaineeThread is empty.

The attribute Direction is two-valued, with possible values `forward` and `backward` according to the direction of the thread-dependency in the system model, as discussed in Section 13.1.3. This information is of course vital if the system model is to be used to find a likely complainee thread. In the current fault isolation scheme, no internal IPC-errors as discussed in Section 13.1.3 are used.

Instance versus class information The information in the error message signature can be divided into two parts; object and thread information respectively. The object information is the Complainer and Complainee attributes. The thread information is the sending thread (attribute SourceThread) and, in the case of an IPC-error message, the ComplaineeThread and Direction attributes.

The object information in the signature can be either on the instance- or class-level, i.e., the signature can tell us what specific object instance of a class the complainer and complainee are, or it can only tell us their respective classes, corresponding to the system model in Section 12.1. Likewise for the thread information in the signature, that can specify the specific thread instance in the run-time system, or just the thread type corresponding to an element in $\mathsf{T_C}$ in the system model in Section 12.2. In the current implementation only class-level and thread type-level information is used in the error message signatures.

As mentioned in Section 10.5, the system needs to fulfill the property that the static class structure reflects the run-time object structure well for the system model to be useful for fault isolation. If this property holds, then class information only in the error message signatures will be sufficient for many fault scenarios.

Apart from making the fault isolation algorithms simpler, the main advantage of having no object instance information in the error message signature is that the class part of the signature can be stored in the fault isolation layer rather than in the system itself, and added to the error message a-posteriori. It significantly reduces the effort to implement a fault isolation scheme on an existing system. For this a-posteriori adding of the signature, the error messages need to fulfill the property that the signature is uniquely identified by the message ID. It is hence necessary that each error message is used only by one class, and only for error conditions with identical (class part) signature.

The base graph In Section 13.4, we will introduce the formal representation of a fault scenario in the form of a so-called base graph. In short, the base graph has the objects and threads that has manifested themselves in the fault scenario as nodes, and the corresponding error messages as edges. How to use the error message signatures to form the base graph should be intuitively clear, and is trivial given the complete base graph definition in Section 13.4.

An instantiated object is described in the the system model (Chapter 12) with possibly several classes in an inheritance hierarchy. The object collaborations in the system are described in the model with associations between classes in an inheritance hierarchy. To be able to use the system model for fault isolation, as will be described in Chapter 14, the formal representation of the complainer and complainee objects in the base graph, needs to include also the applicable inheritance relationships. That the error message signature currently contains only class-level information strengthens this requirement. To this end, we will use the term *generalized object* in Section 13.4 for an object/class identified by an error message signature (attributes Complainer and Complainee in Figure 13.1).

The thread part of the error message signature is somewhat more problematic than the class part. Above, we made the assumption that the class part of the signature can be added a-posteriori, i.e., that an error message can be linked to specific complainer and complainee classes. Even if this property holds, many of those classes may execute on several different threads during run-time. It is hence not always possible to add the thread information to an error message a-posteriori, since not even the type of the sending thread (see Section 11.2.5) might be known.

To resolve this issue and keep the advantage that the error message signature can be added a-posteriori, it is assumed in the current implementation that for the internal and relational error messages (classes Internal and Relational), the sending thread-type is known (attribute SourceThread), but the information is used only to explain IPC-error messages. The fault isolation scheme is then not especially sensitive against missing thread information in the non-IPC error messages. For the IPC-error messages (class IPC) it is, in the ABB Robotics application, possible to get the thread-type information (attributes SourceThread and SendingTread) a-posteriori. The thread view of the fault scenario is further discussed in Section 13.4.

13.2.2 Operational and status messages

When certain important events take place in the system, e.g.,, like major state changes from "running"-mode to "failure", the operator is often informed by a message sent to the system log. These state change messages are called *operational*. There is usually also a number of other *status* messages that are intended mainly as information to the user about normal procedures and events in the system. A number of such operational and status messages cannot be avoided when the system shuts down, see, e.g., the example in Section 10.3.

These operational and status messages are though mostly irrelevant for the fault isolation. As described in Section 13.1.1; an emergency stop is orchestrated from above and will give rise to several operational and other status messages as the system shuts down. These messages contain basically no information useful for the fault isolation scheme in its present form, and are disregarded in the current implementation. The fault isolation scheme is focused on the "real" error messages, discussed in Section 13.2.1, emanating from deviations from normal behavior detected by objects close to the real fault.

In the fault isolation scheme, the operational and status messages are hence only used for clustering the log, see Section 13.3. If the fault isolation approach is extended to take more dynamic behavior into account, the operational and status messages can probably be utilized, though. This is a matter of ongoing and future research.

13.2.3 Supervisory messages

A category of log-messages we will call *supervisory* are sent by safety and supervisory modules of the system when they detect a fault or a fault is reported to them. With careful system design and/or modeling, the messages can be included in the fault isolation scheme in different ways.

Safety modules in the system often maintain so called safety chains, that basically are a number of conditions that must be fulfilled for the system to be allowed to operate normally, e.g., the emergency button is not pressed, safety gates are closed etc. When such a condition is violated, the user needs to be informed, and in our context this means that the safety module sends a log-message.

These messages can primarily be used to cluster the log into fault scenarios, as described in Section 13.3. The potential usefulness of one of these messages in the analysis of a fault scenario, as part of the base graph in Section 13.4, must be settled on a case to case basis. If the system model contains the necessary dependencies, a safety/supervisory message can be classified as relational unknown and then be explained by the base graph extension mechanisms in Chapter 14.

The supervision software could, e.g., consist of analytical redundancy fault detection methods for supervision of hardware, or explicit supervision of certain threads, as mentioned in Section 13.1.3. These messages can be designed and classified as internal and contain information very useful to the user.

13.2.4 Unobservable faults

With *unobservable faults*, we mean that when the fault occurs, and eventually causes the system to halt, there is no directly corresponding error message, i.e., no error message that is most relevant in a natural way.

Faults can be said to be unobservable for at least two different reasons. First, the system may be configured in such a way that certain error messages are deliberately not sent, with the purpose of not cluttering the log. The information that the failure of a certain component is non-observable is then available in the configuration database (see Section 10.4), and could in principle be included in the fault isolation procedure. The current implementation does not take the configuration into account, though.

Second, there might simply not exist an error message directly corresponding to the occured fault, or even an mirror-object corresponding to the equipment it occurred in. In this case, a more relevant term would perhaps be *unknown faults*. The current implementation of *DrRobot* assumes all faults known and represented by relevant error messages. A possible extension is to include a model of the environment, i.e., the hardware and external equipment, the faults that can occur there, and the hardware induced dependencies in the system.

13.3 Clustering the log

As already mentioned, from the viewpoint of an automated fault isolation scheme, the error messages are the observable effects of a fault scenario. Most messages corresponding to a fault scenario arrive in the system log within a few seconds, when the shutdown occurs, but important events in the fault scenario may have occured several minutes before. The system log may also contain messages from old fault scenarios and other messages unrelated to the current fault scenario. To perform automated analysis on a single fault scenario, a mechanism to isolate the messages in the system log that correspond to that fault scenario is hence needed. We will call the collection of log-messages that corresponds to a specific fault scenario a *cluster*.

The clustering problem has not been primarily addressed in this work yet, and no automatic clustering algorithm has been implemented. Many clustering methods can be envisioned, but most will be based on detection of certain events and distance in time between messages.

A clustering method we propose is based on maximal time span of a cluster called *cluster period* and *divider events*. A divider event either indicates that a cluster, i.e., a fault scenario, ends with this event even if the time period has not expired, or that a new cluster (fault scenario) has started if one is not already active. A typical cluster ending divider event is an operational message informing about a restart of the system. A divider event starting a cluster and hence a fault scenario is typically a runtime error message (Section 13.2.1). Each cluster is identified and passed on for analysis.

Design parameters for clustering are the events in the divider group and the length of the cluster period. An adaptive scheme could be considered with, e.g., increasingly larger (or smaller) cluster periods or concatenation of clusters, if the analysis is inconclusive. In the remainder of this thesis, we will assume that the message log can be clustered properly into fault scenarios.

13.4 Base Graph

In this section, we will define and discuss the formal representation of a fault scenario, called a *base graph*. The base graph is used to represent the information from the error messages about the occured error condition propagation in the fault scenario. In Chapter 14 we will show how to use the system model to compensate for incomplete information in the error messages by extending the base graph, and how to use the base graph to find a cause-effect relation on the error messages. The base graph constructed solely from error messages will be called the *initial base graph*.

The base graph is a bit more complicated than an ordinary directed graph, in that it contains three different kind of nodes and three different kind of edges, with certain relationships to each other. The base graph can be considered to consist of two parts, an object part and a thread part, that will be formally defined in Section 13.4.3 The run-time system contains many instantiated objects and threads. A base graph for a specific fault scenario has as nodes the objects and threads that has manifested themselves through the error messages, and the error messages as edges. The base graph hence bears a strong resemblance to UML object and task diagrams.

In the current implementation of the fault isolation scheme and the tool *DrRobot* used for the ABB Robotics application, it is assumed that the sending object of a relational error message might know only the superclass of the actual complainee. Also, the class method that detects the error condition and sends the error message might be defined in a superclass to the instantiated objects class. As discussed in Section 13.2.1, the currently implemented fault isolation scheme uses only class information and no instance information. Hence an instantiated object of the specific class c might manifest itself in the base graph with any class in the set $\mathrm{asc}(\mathsf{c})$ (notation from Chapter 12).

We may hence get several nodes in the base graph corresponding to the same instantiated object in the run-time system. We call these nodes *generalized objects*. It is then also necessary to include the corresponding part of the inheritance relation $\multimap$ from Chapter 12 in the base graph. The following assumption is needed for the generalized objects to make sense, and to justify the use of the system model for base graph extension in Chapter 14.

Assumption 13.1 *For each equivalence class* $[\mathsf{c}] \subseteq \mathsf{Classes}$*, there is at most one instantiated object in the run-time system that manifests itself in a specific fault scenario.*

Assumption 13.1 gives us a design rule for the system model to be useful for fault isolation, that was mentioned also in Section 10.5: Class hierarchies should be as specific as possible, meaning that too general superclasses should be avoided. The assumption of course places a tough formal restriction on the system and, specifically, on the system model. In our experience with the ABB Robotics industrial control system, the assumption, and the system model design rule, are very natural and have always been fulfilled in practice.

The base graph will be presented and discussed using three representations:

- A graphical notation suitable for intuitive understanding, Section 13.4.1
- A data structure suitable for implementation, Section 13.4.2
- A formalism used to state algorithms and prove properties, Section 13.4.3

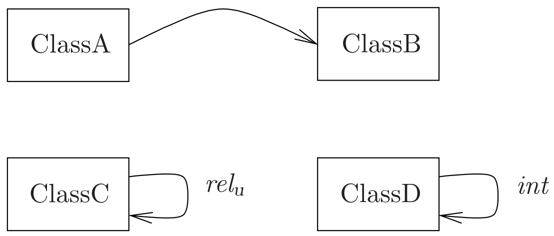


Figure 13.2: Graphical base graph notation for relational, relational unknown and internal error messages (see Table 13.1).

To analyze the base graph, and to decide if and how the base graph should be extended in Chapter 14, we will introduce classifications of the base graph nodes based on how the nodes are connected to each other in the graph. This analysis is introduced in Section 13.4.4.

13.4.1 Graphical notation

The best understanding of the base graph is probably given by the graphical notation. The base graph is visualized with two views, an object view and a thread view. Each view can contain information from both the object and thread part of the base graph (defined formally in Section 13.4.3 below), only differently visualized.

In Figure 13.2 we show the object view graphical notation, i.e., the Relational and Internal error messages (Table 13.1, Section 13.2.1) and their corresponding complainer and complainee objects. A generalized object is simply shown as a rectangle, usually with the class name in it. A relational error message with known complainee is graphically denoted as an arrow between two (generalized) objects, and the relational error message with unknown complainee is denoted by an arrow to self, adorned with rel_u. An internal error message is denoted by an arrow to self adorned with *int*. The complainer can be said to have no complainee, or that the object itself is its own complainee. The semantic for all arrows (edges) in the base graph is "complains on". An internal error message is a complaint from an object on itself, and a relational unknown is, of course, a complaint on an unknown object. Cf. the associations and thread dependencies in a UML model, Chapter 11.2, which have the semantic "uses".

The objects are partitioned into packages in the UML system model, as described in Section 11.2.3, and we show this with a dashed line surrounding the objects contained in a package, and the name of the package at the top left corner.

Example 13.1 Field bus fault scenario
The fault scenario is similar to the fault scenario in Section 10.3. The ABB robot control system is running a user-defined program, when a field-bus unit suffers a fault.

The clustered system log together with the corresponding base graph is shown in Figure 13.3. The error condition propagation in the system can be clearly seen in the base graph, from the bottom and up. Ignore the notation at the lower right corner of the packages for now. It will be explained in Section 13.4.4.

Note the first and last message in the log cluster. They are operational messages (Section 13.2.2), and can serve as divider events as described in Section 13.3. The “real” run-time error messages as described in Section 13.2.1 are the messages numbered 8-12. Their respective signature is made clear in the base graph.

The figure also contains an example of an instantiated object that has manifested itself as two generalized objects, (`RealInstruction` and `rlio`). The inheritance relation is visualized as in the UML, as an arrow with a triangle shaped head. □

As for classes, no instance information for threads is present in the base graph in the current implementation, and we hence need to make a similar assumption as we did for objects in Assumption 13.1.

Assumption 13.2 *For each thread type in the system model, at most one instantiated thread manifests itself in a specific fault scenario.*

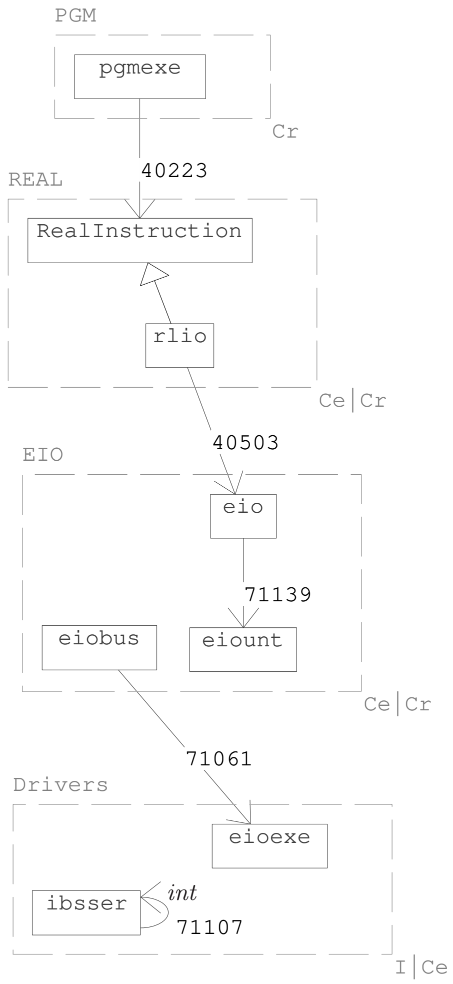
As for the object part of the base graph, the assumption is also needed to justify the use of the system model when extending the graph with the methods in Chapter 14. As Assumption 13.1, the above assumption places a formal restriction on the system, but in our experience so far, this has not been a problem in practice.

A thread (IPC) error message (see Table 13.1 and Figure 13.1) is, as all error messages, sent by an object. In Section 13.2.1, Table 13.1, the thread error messages are divided into three types; relational and relational unknown with direction forward (server) or backward (client). In the object view of the base graph, these three types of thread error messages are denoted as an arrow to self, adorned with $relT$, $relT_u^S$ and $relT_u^C$ respectively. See Figure 13.4(a) and 13.5(a).

Since each message is sent by an object, the message arrow in the object view is adorned with the sending thread, as is shown in Figure 13.4(a). For the relational IPC-error messages, both the complainer and complainee thread is shown. As discussed in Section 13.2.1, thread information is not always available in the current implementation, since error message signatures are added a-posteriori. Hence the sending thread of an ordinary error message is not always present in the base graph.

The thread part of the base graph is visualized as a separate view, closely resembling UML task diagrams. The threads that have manifested themselves in the fault scenario are shown as skew rectangles, and the three types of IPC-error messages are shown as arrows analogous to the object part, adorned with the sending object. The relational unknown IPC-error messages are arrows to self adorned with $relT_u^C$ or $relT_u^S$, and an relational IPC-error message is shown as an arrow between two threads. See Figure 13.4(b) and 13.5(b).

Example 13.2 IPC fault scenario
The fault scenario in this example is from a real situation, where the control system




```
7.  10008 Program restarted 0105 13:45.9
The task MAIN has
restart to execute.
The originator is the production window.

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

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

10.  71139 Access error from IO 0105 13:45.35
Description\Reason:
- Cannot Read or Write signal DO3_1
  due to communication down.

11.  40503 Reference error 0105 13:45.35
Device descriptor is
not valid for a digital write operation

12.  40223 Execution error 0105 13:45.35
Task MAIN: Fatal runtime
error

13.  10020 Execution error state 0105 13:45.35
The program execution has reached
a spontaneous error state

14.  10005 Program stopped 0105 13:45.35
The task MAIN has
stopped. The reason is that
an external or internal stop has
occurred.
```

Figure 13.3: The base graph and system log from the real fault scenario in Example 13.1.

is operating normally, executing a user-defined program. The names of classes, packages and threads have been changed, not to obscure the principle.

What happened was that the thread called Task 3 got hung up in a badly configured communication with an external device, and two other dependent threads, Task 1 and Task 2, failed after trying to contact Task 3. After clustering and filtering of operational messages in the log, only three error messages remained. Task 3 is silent, and only shows up in the base graph because it is the complainee of two relational IPC-error messages.

The base graph for the fault scenario with both object and thread part is shown in Figure 13.4. Since the cluster only consists of three messages, the base graph is quite small. Using the assumption in Section 13.1.2, that error conditions can propagate within packages, the natural conclusion from the base graph is that the problem lies with Task 3. In Section 14.3, we will show how this conclusion is formally reached, and the example is continued in Chapter 15, Example 15.2.

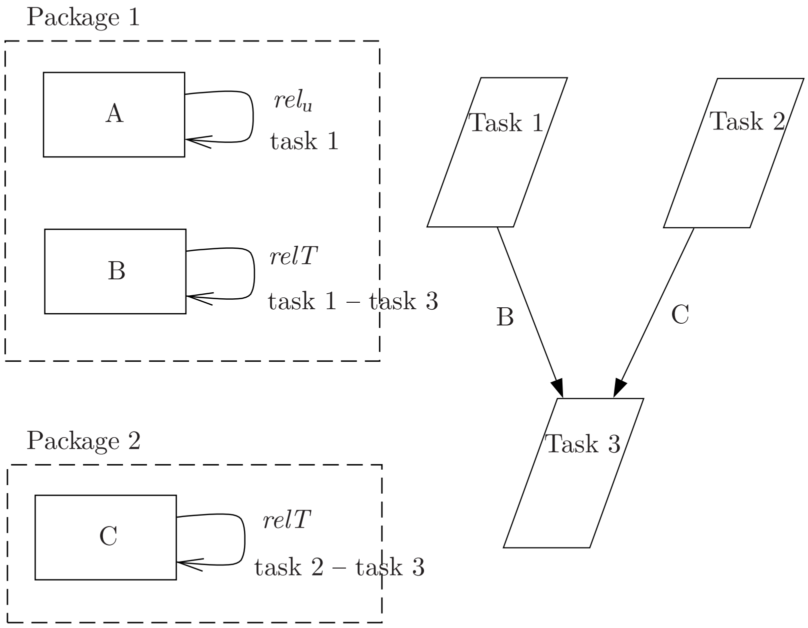
This conclusion is as far as *DrRobot* can go, but since the original error messages as shown in Figure 13.4(c) were quite cryptic, this information strongly focuses the further fault isolation. □

Example 13.3 Real time fault scenario
The scenario here is that during normal operation of the system, one thread feeds another with data according to the enqueue-pattern (see Section 11.2.1).

The fault is that the data-providing thread, the client in the system model, cannot provide data for some reason, e.g., because too high computational load. The result is a fault scenario where the server thread in the enqueue-pattern fails due to data starvation and sends a relational unknown IPC-error message with direction backward (client). After clustering the log, two error messages remain, one relational unknown and the IPC-error message. The base graph is shown in Figure 13.5. In Example 15.3, Chapter 15, we will use the methods introduced in Chapter 14, and show how both the object part and thread part of this base graph can be extended using the system model, and how this extension can be used to get a clearer interpretation of the fault scenario. □

13.4.2 Base graph data structure

We have now informally presented all elements of the base graph. The exact relationships between the graph elements are perhaps most compactly illustrated by the base graph data structure used by the implementation. It is shown in UML class diagram notation in Figure 13.6. As can be seen in the figure, the base graph has three different kinds of nodes, one for (generalized) objects, packages and threads respectively, and three different kind of edges, one for each kind of link that has manifested itself in the fault scenario. The suffix -Link for the base graph edge classes has been chosen after the usual OO-terminology for an instantiated association. As already mentioned in Section 13.4.1, the semantic for an edge in a base graph is different than for a link in an object diagram, though. The normal

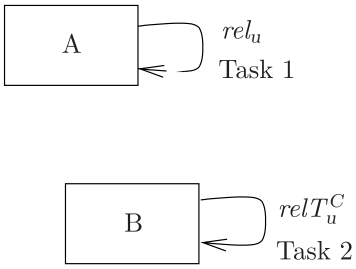


(a) The object part (b) The thread part

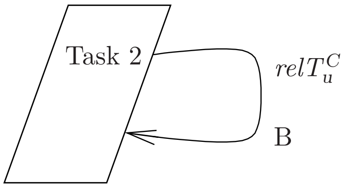
```
27.  71156 IPC queue full 0102 03:36.26
Description\Reason:
- The ipc queue was full, when sending to trap routine.
  errno = 3d0002

1. 90001 safdev.c 211 IPC distribute error -9 0102 03:37.4
safdev.c 211 IPC distribute error -9
2. 90002 safevtts.c 678 internal error response 0102 03:37.4
safevtts.c 678 internal error response
```

(c) The clustered system log

Figure 13.4: The object and thread parts of the base graph for the fault scenario in Example 13.2.



(a) The object part



(b) The thread part

Figure 13.5: The object and thread parts of the base graph for the fault scenario in Example 13.3

semantic for a link in an object diagram is “uses”, but an edge in the base graph has the semantic “complains”, or “complains on”.

The error messages classified as Internal and Relational in Table 13.1, Section 13.2.1 are represented by edges of type ObjectLink, and the IPC-error messages are represented by the edge-type ThreadLink. The packages are just a partition of the objects as described in Section 11.2.3. The package-links, represented in Figure 13.6 with PackageLink, that are present in the initial base graph are all induced by corresponding object links between objects in different packages, as will be further described below.

The attributes for the base graph classes correspond to the attributes in the error message data structure in Figure 13.1. The method NodeCl(), specified for the base graph nodes in Figure 13.6(b), will compute the classification of the node, to be defined in Section 13.4.4.

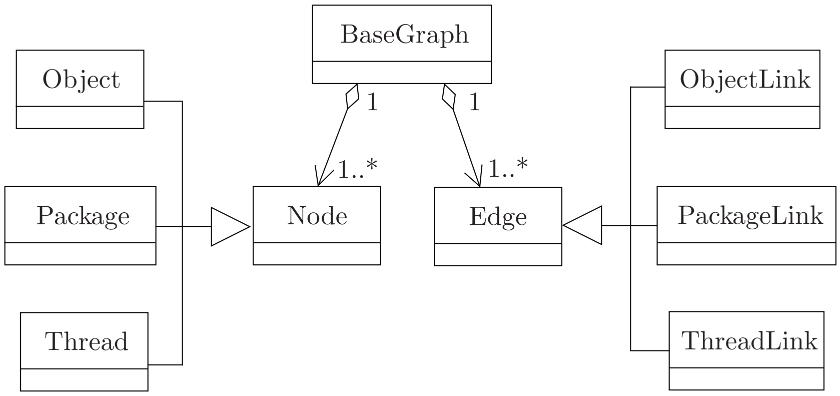
13.4.3 Formal representation

For the purpose of stating algorithms and proving properties in Chapter 14, we will in this section introduce a formal representation of the base graph, and some other definitions in connection with the base graph. The class diagram over the base graph data structure in Figure 13.6(b) could be useful to have in mind.

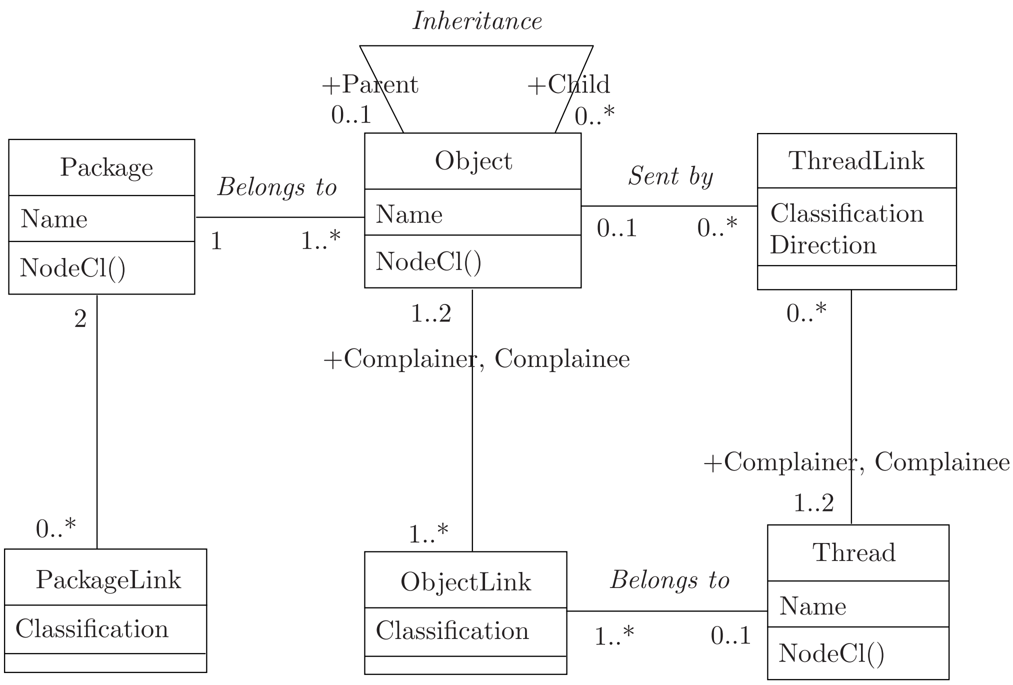
By comparing with Figure 13.6(b), it should not come as a surprise that a base graph formally consists of a six-tuple

$$< \mathsf{O}, \mathsf{P}_\mathsf{O}, \mathsf{T}_\mathsf{O}, \mathsf{OL}, \mathsf{PL}, \mathsf{TL} > . \tag{13.1}$$

All six elements will be defined and explained below. The first three elements are



(a) The base graph elements.



(b) Detailed relationships between the base graph elements.

Figure 13.6: The Base Graph data structure.

the sets

$$\begin{array}{ll} \mathsf{O} & \text{— generalized objects} \\ \mathsf{P_O} & \text{— packages (a partition of } \mathsf{O}) \\ \mathsf{T_O} & \text{— threads} \end{array}$$

Each generalized object $\mathsf{o} \in \mathsf{O}$ belongs to exactly one class $\mathsf{c} \in \mathsf{Classes}$ in the system model defined in Chapter 12. We define the mapping class : $\mathsf{O} \longrightarrow \mathsf{Classes}$ as

$$\text{class}(\mathsf{o}) = \mathsf{c} \in \mathsf{Classes}.$$

Under Assumption 13.1, the mapping is injective.

The base graph package partition $\mathsf{P_O}$ is defined by the system model package partition $\mathsf{P_C}$. For $\mathsf{o}_1, \mathsf{o}_2 \in \mathsf{O}$, we have

$$\mathsf{o}_1, \mathsf{o}_2 \in \mathsf{p_o} \in \mathsf{P_O} \text{ iff } \text{class}(\mathsf{o}_1), \text{class}(\mathsf{o}_2) \in \mathsf{p_c} \in \mathsf{P_C}.$$

It follows immediately that each partition in $\mathsf{P_O}$ corresponds to a unique package in $\mathsf{P_C}$. With $\mathsf{p_c}$ and $\mathsf{p_o}$ as above, we define the mapping pmap : $\mathsf{P_O} \longrightarrow \mathsf{P_C}$ as

$$\text{pmap}(\mathsf{p_o}) = \mathsf{p_c}.$$

This mapping is of course always injective, regardless of Assumption 13.1.

Each thread $\mathsf{t_o}$ that has manifested itself in the fault scenario, i.e., $\mathsf{t_o} \in \mathsf{T_O}$, corresponds to exactly one thread $\mathsf{t_c} \in \mathsf{T_C}$ in the system model. We define the mapping tmap : $\mathsf{T_O} \longrightarrow \mathsf{T_C}$ as

$$\text{tmap}(\mathsf{p_o}) = \mathsf{p_c} \in \mathsf{T_C}.$$

Under Assumption 13.1, the mapping is injective.

The three *link sets*, or *edge sets* in the base graph six-tuple (13.1) are

$$\begin{array}{rclll} \mathsf{o}_1 \overset{\mathsf{c,t}}{\to} \mathsf{o}_2 \in & \mathsf{OL} & \subset & (\mathsf{O} \times \mathsf{C_{OL}} \times \mathsf{T_O} \times \mathsf{O}) & \text{— Object links} \\ \mathsf{p}_1 \overset{\mathsf{c}}{\to} \mathsf{p}_2 \in & \mathsf{PL} & \subset & (\mathsf{P_O} \times \mathsf{C_{PL}} \times \mathsf{P_O}) & \text{— Package links} \\ \mathsf{t}_1 \overset{\mathsf{c,o}}{\to} \mathsf{t}_2 \in & \mathsf{TL} & \subset & (\mathsf{T_O} \times \mathsf{C_{TL}} \times \mathsf{O} \times \mathsf{T_O}) & \text{— Thread links.} \end{array}$$

The complete collection of *link classifications* are

$$\begin{array}{lcl} \mathsf{C_{OL}} & = & \{\, int, rel, rel_u, rel_{\mathrm{der}} \,\} \\ \mathsf{C_{PL}} & = & \{\, obj, pack \,\} \\ \mathsf{C_{TL}} & = & \{\, relT, relT_u^C, relT_u^S, relT_{\mathrm{der}}, intT_{\mathrm{der}} \,\} \end{array} \tag{13.2}$$

All classifications except rel_{der}, $pack$, $relT_{\mathrm{der}}$ and $intT_{\mathrm{der}}$ have been discussed in Section 13.2.1. For the object and thread links with internal or relational unknown classifications, of course only the complainer object or thread is interesting. For simplicity of notation, we will make the complainee the same as the complainer if we have to specify, i.e., for the object links, $\mathsf{o}_1 \overset{rel_u,\mathsf{t}}{\to} \mathsf{o}_1$ and $\mathsf{o}_1 \overset{int,\mathsf{t}}{\to} \mathsf{o}_1$.

The *relational derived* classifications, rel_{der} for object links, $relT_{\mathrm{der}}$ for thread links and *pack* for package links are new classifications. They are used to denote edges added to the base graph using the system model, as will be further explained in Chapter 14.

The *internal derived* classification for thread links, $intT_{\mathrm{der}}$, is used only when constructing the cause-effect relation in Section 14.3. As discussed in Section 13.1.3, a thread with problems often is silent, and if a silent thread in the base graph has more than one complaint on it from other threads, we will in Section 14.3 use an extra thread link to represent the internal IPC-error message the silent thread should/could have sent. This extra thread link will not be added to the base graph, though, since that information is not needed to interpret the base graph, and would only clutter the picture (see Example 13.2 and 15.2).

Package links PL are always relational with known complainee. The package links with classification *obj* are induced by a relational object link, classification *rel* or rel_{der}, between objects in different packages. A package link $\mathtt{p}_1 \overset{obj}{\to} \mathtt{p}_2$ is hence a member of PL if and only if there exists an object link $\mathtt{o}_1 \overset{\mathtt{c},\mathtt{t}}{\to} \mathtt{o}_2 \in \mathsf{OL}$ such that

$$\mathtt{c} \in \{\, rel, rel_{\mathrm{der}} \,\},\ \mathtt{o}_1 \in \mathtt{p}_1 \text{ and } \mathtt{o}_2 \in \mathtt{p}_2.$$

The *obj* package links are hence redundant information, but are present as syntactic sugar to make the formulation of the extension algorithms in Chapter 14 easier and to stress the relationship between $< [\mathsf{O}], \mathsf{OL} >$ and $< \mathsf{P_O}, \mathsf{PL} >$. Package links with classification *pack* are added to PL using the system model, as further explained in Chapter 14.

Given the object link $\mathtt{ol} = \mathtt{o}_1 \overset{\mathtt{c},\mathtt{t}}{\to} \mathtt{o}_2 \in \mathsf{OL}$, we define the selector functions:

$$\begin{array}{lcll}
\mathrm{cr}(\mathtt{ol}) & = & \mathtt{o}_1 & \text{— complainer} \\
\mathrm{ce}(\mathtt{ol}) & = & \mathtt{o}_2 & \text{— complainee} \\
\mathrm{cl}(\mathtt{ol}) & = & \mathtt{c} & \text{— object link classification} \\
\mathrm{th}(\mathtt{ol}) & = & \mathtt{t} & \text{— sending thread.}
\end{array}$$

Corresponding selector functions for the thread link $\mathtt{tl} = \mathtt{t}_1 \overset{\mathtt{c},\mathtt{o}}{\to} \mathtt{t}_2$ are

$$\begin{array}{lcll}
\mathrm{cr}(\mathtt{tl}) & = & \mathtt{t}_1 & \text{— complainer (i.e., sending) thread} \\
\mathrm{ce}(\mathtt{tl}) & = & \mathtt{t}_2 & \text{— complainee thread} \\
\mathrm{cl}(\mathtt{tl}) & = & \mathtt{c} & \text{— thread link classification} \\
\mathrm{obj}(\mathtt{tl}) & = & \mathtt{o} & \text{— sending object.}
\end{array}$$

The selector functions for a package link $\mathtt{pl} = \mathtt{p}_1 \overset{\mathtt{c}}{\to} \mathtt{p}_2$ are then of course

$$\begin{array}{lcll}
\mathrm{cr}(\mathtt{pl}) & = & \mathtt{p}_1 & \text{— complainer package} \\
\mathrm{ce}(\mathtt{pl}) & = & \mathtt{p}_2 & \text{— complainee package} \\
\mathrm{cl}(\mathtt{pl}) & = & \mathtt{c} & \text{— package link classification}
\end{array}$$

The base graph is initially built up using error messages from a cluster of the system log. Each error message correspond, in the obvious way, to an object link

$\mathsf{o}_1 \overset{\mathsf{c,t}}{\rightarrow} \mathsf{o} \in \mathsf{OL}$ where $\mathsf{c} \in \{\, int, rel, rel_u \,\} \subset \mathsf{C_{OL}}$, or a thread link $\mathsf{t}_1 \overset{\mathsf{c,o}}{\rightarrow} \mathsf{t}_2 \in \mathsf{TL}$ where $\mathsf{c} \in \{\, relT, relT_u^C, relT_u^S \,\} \subset \mathsf{C_{TL}}$. We let M be the set of all links corresponding to error messages involved in a base graph, that is, $\mathsf{M} \subseteq \mathsf{OL} \cup \mathsf{TL}$. Initially we have $\mathsf{M} = \mathsf{OL} \cup \mathsf{TL}$.

The inheritance relation for classes extends to generalized objects in the natural manner:

$$\mathsf{o} \twoheadrightarrow \mathsf{o}' \text{ iff } \mathrm{class}(\mathsf{o}) \twoheadrightarrow \mathrm{class}(\mathsf{o}')$$

where $\mathsf{o}, \mathsf{o}' \in \mathsf{O}$. We also impose the property on the base graph that generalized objects corresponding to the classes directly between class(o) and class(o') in the inheritance tree are present in the base graph. This is mainly for clarity of visualization of the base graph.

The function rel($\cdot$) from Chapter 12 also extends to generalized objects:

$$\mathrm{rel}(\mathsf{o}) = \{\, \mathsf{o}' \in \mathsf{O} \mid \mathsf{o} \twoheadrightarrow \mathsf{o}' \text{ or } \mathsf{o}' \twoheadrightarrow \mathsf{o} \,\}$$

and given the reflexive and symmetric closure of the inheritance relation, $[\mathsf{o}]$ denotes the equivalence class that contains o. We also denote the set of equivalence classes in O as $[\mathsf{O}]$. Assumption 13.1 implies that $[\mathsf{o}] = \mathrm{rel}(\mathsf{o})\ \forall \mathsf{o} \in \mathsf{O}$, and that all generalized objects in $[\mathsf{o}]$ correspond to the same instantiated object. Hence, the object link set OL is also interpreted as a relation over $[\mathsf{O}]$. The elements in $[\mathsf{O}]$ will simply be referred to as *objects*.

From the base graph six-tuple

$$< \mathsf{O}, \mathsf{P_O}, \mathsf{T_O}, \mathsf{OL}, \mathsf{PL}, \mathsf{TL} > .$$

we can form three subgraphs, that each trivially is an ordinary directed graph:

$$< [\mathsf{O}], \mathsf{OL} >, < \mathsf{P_O}, \mathsf{PL} > \text{ and } < \mathsf{T_O}, \mathsf{TL} > . \tag{13.3}$$

The first two subgraphs constitute the object part of the base graph, and the last subgraph the thread part.

The object part of the (initial) base graph is hence hierarchically built up by the graph $< [\mathsf{O}], \mathsf{OL} >$ and the overlaid graph $< \mathsf{P_O}, \mathsf{PL} >$. An important property is that even if $< [\mathsf{O}], \mathsf{OL} >$ is not connected, $< \mathsf{P_O}, \mathsf{PL} >$ might be. During the package extension described in Section 14.2.2, the *pack* package links are added only between unconnected parts of $< \mathsf{P_O}, \mathsf{PL} >$ and hence $< [\mathsf{O}], \mathsf{OL} >$.

Some graph theory For use in the sequel, we define some graph theory concepts for directed graphs. We assume that the reader is familiar with the basic concepts of graph theory, but for a thorough treatment see, e.g., [73].

Definition 13.5 Connected graph and connected component
An undirected graph is called *connected*, if there exists a path between any two nodes in the graph.

A directed graph is called *connected*, if its symmetric closure, which is an undirected graph, is connected.

A *connected component* in a graph is a set of nodes forming a connected subgraph, such that no larger connected subgraph involving the same nodes exists. □

Note that a graph that is connected consists of exactly one connected component.

Definition 13.6 Loops and cycles in directed graphs
We define a *loop*, or a *cycle*[2], in a directed graph as a set of edges, ignoring edges to self, that lead from a node, back to the same node. A directed graph without loops is called *acyclic*. □

13.4.4 Node classifications

In this section, we will introduce some analysis tools for the base graph, for further use in Chapter 14. As will be described in detail below, the nodes in the base graph are classified in terms of the base graph links, i.e., error messages, they participate in.

To briefly explain the rationale behind the node classification, consider for simplicity the object part of the base graph. The properties of the objects that has manifested themselves in the fault scenario that we try to quantify are:

- Does the object have someone to blame for the experienced problems?
- Is the object potentially the closest to the occured fault?

If an object does not have someone else to blame, and is not considered a good candidate for being closest to the actual fault, then we can use the system model to attempt an extension of the base graph, by finding a complainee for the object. This is discussed in detail in Chapter 14.

Object node classification

The elements in $[\mathsf{O}]$ are classified according to their local situation in terms of link participation in the base graph, which is expressed as a function:

$$\mathrm{NodeCl} : [\mathsf{O}] \longrightarrow 2^{\mathsf{C_O}}.$$

Cf. Figure 13.6(b). The *elementary object classifications* are

$$\mathsf{C_O} = \{U, I, Ce, Cr, UT, CrT\}. \tag{13.4}$$

Which elementary object classifications that are applicable to a specific object in the base graph, is decided by the object links and thread links the object participate in, as explained in Table 13.2. The complete *object classification* $\mathrm{NodeCl}(\mathsf{o})$ for

[2]Some authors define a cycle as a loop involving more than two nodes.

$\mathsf{C_O}$	Explanation
U	Object has sent a relational unknown message (rel_u)
I	Object has sent an internal error message (int)
Ce	Object is complainee of a relational link (rel or rel_{der})
Cr	Object is complainer of a relational link (rel or rel_{der})
UT	Object has sent a relational unknown IPC-error message ($relT_u^C$ or $relT_u^S$)
CrT	Object has sent a relational IPC-error message ($relT$)

Table 13.2: Informal explanation of the elementary object classifications in Equation (13.4).

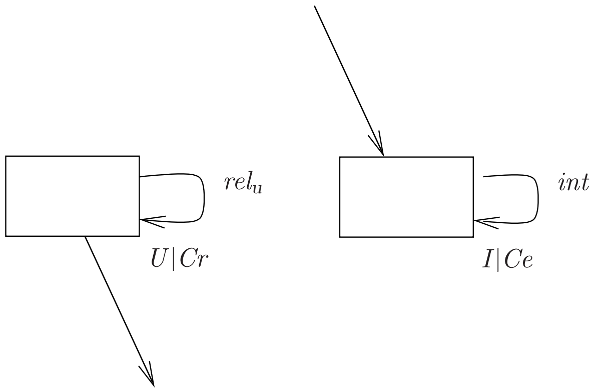


Figure 13.7: Example of object classifications according to Table 13.2.

an object $\mathsf{o} \in [\mathsf{O}]$ hence consists of any combination of the elementary object classifications, i.e., of any combination of the properties in Table 13.2.

Some examples of how the elementary object classifications are combined are shown in Figure 13.7. The object classification is shown at the lower right corner of the object, and we use the infix operator $\cdot|\cdot$ to denote set union.

The formalization of Table 13.2, i.e., the formal definition of NodeCl(o) for $\mathsf{o} \in [\mathsf{O}]$, is made with six rules, one for each elementary object classification ($\mathsf{C_O}$ in Equation (13.4)).

Definition 13.7 Object classification
The *object classification* for a base graph object node in $[\mathsf{O}]$ is given by a function

$$\mathrm{NodeCl} : [\mathsf{O}] \longrightarrow 2^{\mathsf{C_O}}.$$

defined with six monotone rules:

$$
\begin{array}{rll}
U \in \mathrm{NodeCl}(\mathsf{o}) & \text{iff} & \exists \mathsf{ol} \in \mathsf{OL} \text{ s.t. } \mathrm{cl}(\mathsf{ol}) = rel_u \text{ and } \mathrm{cr}(\mathsf{ol}) \in [\mathsf{o}] \\
I \in \mathrm{NodeCl}(\mathsf{o}) & \text{iff} & \exists \mathsf{ol} \in \mathsf{OL} \text{ s.t. } \mathrm{cl}(\mathsf{ol}) = int \text{ and } \mathrm{cr}(\mathsf{ol}) \in [\mathsf{o}] \\
Ce \in \mathrm{NodeCl}(\mathsf{o}) & \text{iff} & \exists \mathsf{ol} \in \mathsf{OL} \text{ s.t. } \mathrm{cl}(\mathsf{ol}) \in \{\, rel, rel_{\mathrm{der}} \,\} \text{ and } \mathrm{ce}(\mathsf{ol}) \in [\mathsf{o}] \\
Cr \in \mathrm{NodeCl}(\mathsf{o}) & \text{iff} & \exists \mathsf{ol} \in \mathsf{OL} \text{ s.t. } \mathrm{cl}(\mathsf{ol}) \in \{\, rel, rel_{\mathrm{der}} \,\} \text{ and } \mathrm{cr}(\mathsf{ol}) \in [\mathsf{o}] \\
UT \in \mathrm{NodeCl}(\mathsf{o}) & \text{iff} & \exists \mathsf{tl} \in \mathsf{TL} \text{ s.t. } \mathrm{cl}(\mathsf{tl}) \in \{\, relT_u^C, relT_u^S \,\} \text{ and } \mathrm{obj}(\mathsf{tl}) \in [\mathsf{o}] \\
CrT \in \mathrm{NodeCl}(\mathsf{o}) & \text{iff} & \exists \mathsf{tl} \in \mathsf{TL} \text{ s.t. } \mathrm{cl}(\mathsf{tl}) = relT \text{ and } \mathrm{obj}(\mathsf{tl}) \in [\mathsf{o}]
\end{array}
$$

□

An object classification in the base graph hence consists of any element from $2^{\mathsf{C_O}}$. Note that no object in a base graph can have the empty classification, since all objects in the base graph have manifested themselves as participants in at least one link. The number of possible object classifications is hence $|2^{\mathsf{C_O}}| = 2^6 - 1 = 63$. If the object classifications corresponding to IPC-errors are left out, there is only $2^4 - 1 = 15$ object classifications.

Many of the possible object classifications represent undesired and ill-designed behavior for an object. We will define the set of *well formed* object classifications, based on a pragmatic and common sense view on how an object should and should not react to error conditions, with the focus on what is needed from a fault isolation point of view. The partition of object classifications, and hence object behavior, into well formed and non-well formed, constitutes a design principle for system designers and implementors.

The main guiding principle when defining well formedness, is that an object should send only one error message in a given fault scenario, which is a sound principle in object oriented design in general. Here we have chosen not to consider several error messages with the same classification from one single object as non-well formed, since that gives no formal problems for the fault isolation scheme. Such object behavior is most often also undesired, though, and might, e.g., lead to inconclusive results from the fault isolation scheme if the same object has sent several internal error messages.

For simplicity of notation when defining well formedness, we will use the symbol T as a "wildcard", that can be replaced with either of the elementary object classifications UT and CrT(see Table 13.2). The following subset of $2^{\mathsf{C_O}}$ will be called the *well formed* object classifications:

$$\mathsf{WF}_{\mathsf{C_O}} = \{\, U, U|Ce, Ce, I, I|Ce, Cr, U|Cr, U|Ce|Cr, Ce|Cr, T, Ce|T \,\} \tag{13.5}$$

where, e.g., U is interpreted as $\{U\}$ and the infix operator $\cdot|\cdot$ is used to denote set union. In Figure 13.8, the graphical notation for all well formed objects, except the ones involving IPC-error messages, are shown. The well formed object classifications involving IPC-errors are shown graphically in Figure 13.9.

Object classifications from $2^{\mathsf{C_O}} \setminus \mathsf{WF}_{\mathsf{C_O}}$ are called *non-well formed*. A good insight in the intent behind the choice of partitioning into well-formed and non-well-formed is supplied by the fact that the non-well-formed classifications are given

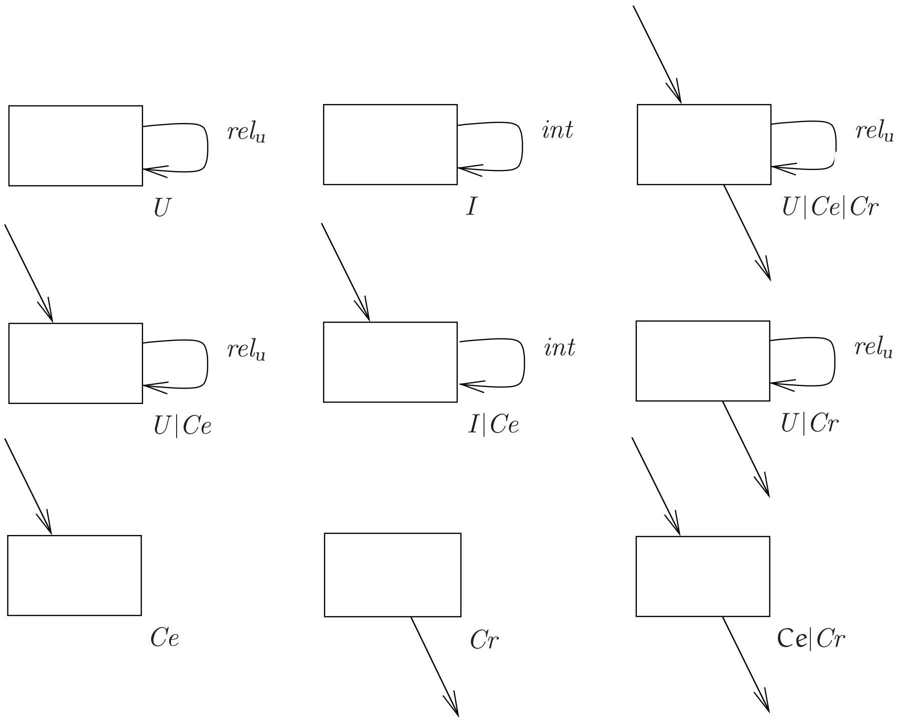


Figure 13.8: The well formed object classifications $\mathsf{WF}_{\mathsf{C_o}}$ from Equation (13.5) (IPC-errors excluded).

by

$$I|Cr, Cr|T, U|I, U|T, I|T \text{ and } UT|CrT \tag{13.6}$$

and exactly all supersets thereof in $2^{\mathsf{C_o}}$, as is easily checked. The classifications in Equation (13.6) will be called the *minimal non-well formed object classifications.* The graphical notation for the two object classifications in Equation (13.6) not involving IPC-errors are shown in Figure 13.10.

An error message should be given the internal classification, corresponding to elementary object classification I, only if it always will be at the root of the software error condition propagation. The IPC-error messages, corresponding to T, are complaints on another thread, known or unknown, and hence cannot be explained further in the object part of the base graph. For that reason, they should also serve as the root of an error condition propagation chain in the object part of the base graph (see Section 13.1.1). The relational error messages, corresponding to Cr and U, are in turn complaints on another object, known or unknown. The minimal non-well formed object classifications in Equation (13.6) are the cases where one of the links that should root an error condition propagation chain (classifications

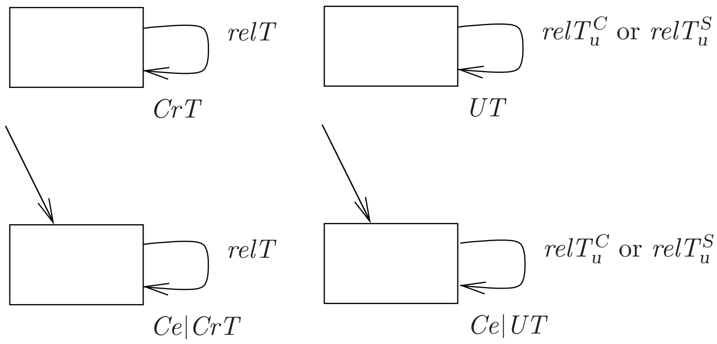


Figure 13.9: The well formed object classifications $\mathsf{WF}_{\mathsf{C_O}}$ from Equation (13.5) involving IPC-errors.

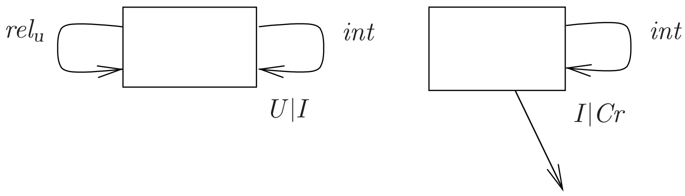


Figure 13.10: The non-well formed object classifications from Equation (13.6) (IPC-errors excluded).

I, UT and CrT) is combined in an object with a link that continues the chain (classifications Cr and U), plus the cases with multiple chain rooting links ($I|T$ and $UT|CrT$).

Other partitionings of the object classifications than the one defined in Equation (13.5) can of course be considered. For example, as just argued above, it should probably be considered undesired system behavior if an object sends both a relational unknown (rel_u) and a relational error message (rel) in the same fault scenario (object classifications $U|Cr$ and $U|Ce|Cr$). The interpretation of these object classifications from a fault isolation point of view is clear for these classifications, though, see Chapter 14. After extension of the base graph using the methods in Chapter 14, objects may get these classifications due to added rel_{der} object links. For simplicity, we have then chosen to consider these classifications as well formed. As a support for designers and implementors, the fault isolation tool can of course easily warn when these object classifications show up in a fault scenario, by distinguishing between the object classification Cr induced by relational (rel) and relational derived (rel_{der}) object links (cf. Table 13.2 and Definition 13.7).

If $U|Cr$ and $U|Ce|Cr$ are considered non-well formed also, it turns out that the set of classifications whose supersets give all non-well formed classifications

$\mathsf{C_P}$	Explanation
U	Object in package has sent a relational unknown error message
I	Object in package has sent an internal error message
Ce	Package is complainee of a package link
Cr	Package is complainer of a package link
UT	Object in package has sent a relational unknown IPC-error message
CrT	Object in package has sent a relational IPC-error message
0	None of the above

Table 13.3: Informal explanation of the elementary package classifications in Equation (13.7).

from Equation (13.6) is simply extended with $U|Cr$. This in turn means that the non-well formed classifications would be characterized exactly as any combination involving at least two of the elementary classifications U, Cr, I, UT and CrT. This is identical to saying that an object should send only one error message in a given fault scenario (ignoring multiple messages with the same link classification).

The reason for non-well formed classifications to occur can be a sign of multiple faults, or, more probable, bad class or module design. The implemented tool currently warns when a non-well formed object is encountered in a base graph, which is intended to be useful for designers and implementors during testing of the system and modeling for the fault isolation scheme.

Package node classification

The packages $\mathsf{P_O}$ and package links PL, belongs to the object part of the base graph, as discussed in Section 13.4.3. Packages in the base graph are, analogous to objects, classified according to their local situation in terms of link participation, expressed as the function:

$$\mathrm{NodeCl} : \mathsf{P_O} \longrightarrow 2^{\mathsf{C_P}}$$

where the *elementary package classifications* are

$$\mathsf{C_P} = \{U, I, Ce, Cr, UT, CrT, 0\}. \tag{13.7}$$

The package classification is given by the link participation of the objects belonging to the package as well as the explicit package links, as is briefly explained in Table 13.3. Recall from Section 13.4.3 that package links can be either induced by relational object links (package link classification *obj*) or explicitly added to the base graph (package link classification *pack*).

The elementary classifications are syntactically identical to the elementary object classifications in the previous section, except for the "empty" classification 0. The object nodes would not exist in the base graph without at least one of the elementary object classifications to apply, but this is not true for package nodes.

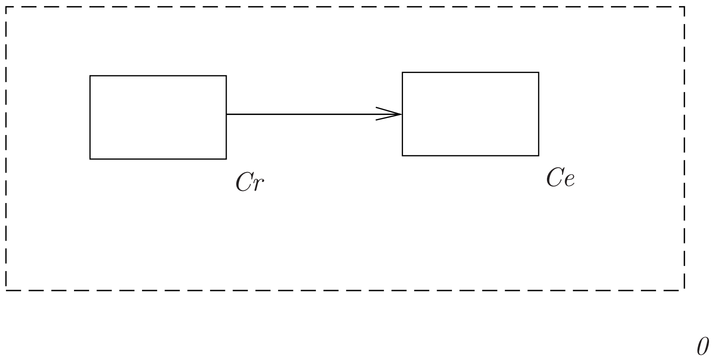


Figure 13.11: How the empty package classification, *0*, in Equation (13.7) and Table 13.3 may occur.

The reason is illustrated in Figure 13.11. The nature of the empty elementary package classification *0* is of course such that it always occurs alone.

The formalization of Table 13.3, i.e., the formal definition of NodeCl($\mathtt{p}$) for $\mathtt{p} \in \mathsf{P_O}$, is made with seven rules, one for each elementary package classification ($\mathsf{C_P}$ in Equation (13.7)).

Definition 13.8 Package classification
The *package classification* for a base graph package node in $\mathsf{P_O}$ is given by a function

$$\mathrm{NodeCl} : [\mathsf{O}] \longrightarrow 2^{\mathsf{C_P}}.$$

defined with seven rules:

$$\begin{array}{rll}
U \in \mathrm{NodeCl}(\mathtt{p}) & \text{iff} & \exists \mathtt{o} \in \mathtt{p}, U \in \mathrm{NodeCl}(\mathtt{o}) \\
I \in \mathrm{NodeCl}(\mathtt{p}) & \text{iff} & \exists \mathtt{o} \in \mathtt{p}, I \in \mathrm{NodeCl}(\mathtt{o}) \\
Ce \in \mathrm{NodeCl}(\mathtt{p}) & \text{iff} & \exists \mathtt{pl} \in \mathsf{PL} \text{ s.t. } \mathrm{ce}(\mathtt{pl}) = \mathtt{p} \\
Cr \in \mathrm{NodeCl}(\mathtt{p}) & \text{iff} & \exists \mathtt{pl} \in \mathsf{PL} \text{ s.t. } \mathrm{cr}(\mathtt{pl}) = \mathtt{p} \\
UT \in \mathrm{NodeCl}(\mathtt{p}) & \text{iff} & \exists \mathtt{tl} \in \mathsf{TL} \text{ s.t. } \mathrm{cl}(\mathtt{tl}) \in \{\, relT_u^C, relT_u^S \,\} \text{ and } \mathrm{obj}(\mathtt{tl}) \in \mathtt{p} \\
CrT \in \mathrm{NodeCl}(\mathtt{p}) & \text{iff} & \exists \mathtt{tl} \in \mathsf{TL} \text{ s.t. } \mathrm{cl}(\mathtt{tl}) = relT \text{ and } \mathrm{obj}(\mathtt{tl}) \in \mathtt{p} \\
0 \in \mathrm{NodeCl}(\mathtt{p}) & \text{iff} & \text{none of the above apply}
\end{array}$$

□

Analogous to the object node classifications, the following subset of $2^{\mathsf{C_P}}$ will be called *well formed* package classifications:

$$\mathsf{WF}_{\mathsf{C_P}} = \{\, U, U|Ce, Ce, I, I|Ce, Cr, U|Cr, U|Ce|Cr, Ce|Cr, T, Ce|T, U|T, U|Ce|T, U|I, U|I|Ce, 0 \,\} \quad (13.8)$$

where the symbol T us used as a "wildcard" that can be substituted for UT, CrT or $UT|CrT$. Package classifications from $2^{\mathsf{C_P}} \setminus \mathsf{WF}_{\mathsf{C_P}}$ are called *non-well formed*.

This partitioning is strongly related to the partitioning of the object classifications into well formed and non-well formed, see $\mathsf{WF}_{\mathsf{C_O}}$ in Equation (13.5). The difference is basically that we have to be more liberal with what is undesired behavior on the package level. Even though the elementary classifications for object nodes and package nodes are defined differently, they are semantically very similar, and it is interesting to note that syntactically we have $\mathsf{WF}_{\mathsf{C_P}} \supset \mathsf{WF}_{\mathsf{C_O}}$, which is why we say the partitioning of the package classifications are more liberal.

The more liberal nature of the partitioning is probably best illustrated using the fact that that the non-well formed package classifications are given by

$$I|Cr,\ Cr|T \text{ and } I|T \tag{13.9}$$

and exactly all supersets thereof in $2^{\mathsf{C_P}}$. These classifications will be called the *minimal non-well formed package classifications*. A direct comparison with the minimal non-well formed object classifications in Equation (13.6) shows that a number of minimal non-well formed classifications are simply removed from Equation (13.6) to Equation (13.9) (the different interpretation of T has no influence, since we are interested in the supersets).

The minimal classifications that are removed from Equation (13.6) to Equation (13.9), are $U|I$, $U|T$ and $UT|CrT$. The first two are a combination of an error condition propagation chain rooting message (classifications I and T), and a further complaint on another object (classification U). Both may occur in the same package, without coming from the same object. The third, $CrT|UT$, is a combination of several IPC-error messages, which might very well occur within a package that is used by several threads.

The two first minimal classifications in Equation (13.9), that hence are considered non-well formed, are when an error condition propagation chain rooting message (classification I and T) is combined with a continuation of the chain to *another package* (classification Cr). We have also made the design decision that the combination internal error message/IPC-error message (classification $I|T$) should be considered non-well formed also on the package level.

How to partition the package classifications, i.e., what is considered desired respectively undesired behavior, is much more a design issue for a specific system than the corresponding partitioning of object node classifications. It depends, e.g., on how fine the package partition is for the system under consideration.

We will call a base graph *well formed* if all elements in O and $\mathsf{P_O}$ have well formed classifications.

Thread node classification

The thread part of the base graph, i.e., $< \mathsf{T_C}, \mathsf{TL} >$, will be slightly differently treated than the object part. The thread design in a system most often does not constitute a logical partitioning of the system, as does the class design, and a thread can have many disparate tasks to fulfill. Currently, the thread part is

$\mathsf{C_T}$	Explanation
UT^C	Thread has sent a relational unknown error message, direction client
UT^S	Thread has sent a relational unknown error message, direction server
CeT	Thread is complainee of a relational thread link ($relT$ or $relT_{\mathrm{der}}$)
CrT	Thread is complainer of a relational thread link ($relT$ or $relT_{\mathrm{der}}$)

Table 13.4: Informal explanation of the elementary thread node classifications in Equation (13.10).

only used for explaining relational unknown IPC-error messages. The complete set of elementary thread node classifications corresponding to the object and package node elementary classifications are not really needed for our purposes, but they are easily defined and we will introduce them for completeness. The issue of well-formedness and what could be regarded as desired/undesired behavior will only be briefly discussed, though.

The elementary thread node classifications are

$$\mathsf{C_T} = \{\, UT^C, UT^S, CeT, CrT \,\}. \tag{13.10}$$

and a brief explanation is given in Table 13.4. Analogous to objects and packages, the thread node classification is given by the function

$$\mathrm{NodeCl} : \mathsf{T_O} \longrightarrow 2^{\mathsf{C_T}},$$

and the formal definition of $\mathrm{NodeCl}(\mathtt{t})$ for a thread node $\mathtt{t} \in \mathsf{T_O}$ is given by the rules

$$\begin{array}{lll}
UT^C \in \mathrm{NodeCl}(\mathtt{t}) & \text{iff} & \exists \mathtt{tl} \in \mathsf{TL} \text{ s.t. } \mathrm{cl}(\mathtt{tl}) = relT_u^C \text{ and } \mathrm{th}(\mathtt{tl}) = \mathtt{t} \\
UT^S \in \mathrm{NodeCl}(\mathtt{t}) & \text{iff} & \exists \mathtt{tl} \in \mathsf{TL} \text{ s.t. } \mathrm{cl}(\mathtt{tl}) = relT_u^S \text{ and } \mathrm{th}(\mathtt{tl}) = \mathtt{t} \\
CeT \in \mathrm{NodeCl}(\mathtt{t}) & \text{iff} & \exists \mathtt{tl} \in \mathsf{TL} \text{ s.t. } \mathrm{cl}(\mathtt{tl}) \in \{\, relT, relT_{\mathrm{der}} \,\} \text{ and } \mathrm{ce}(\mathtt{tl}) = \mathtt{t} \\
CrT \in \mathrm{NodeCl}(\mathtt{t}) & \text{iff} & \exists \mathtt{tl} \in \mathsf{TL} \text{ s.t. } \mathrm{cl}(\mathtt{tl}) \in \{\, relT, relT_{\mathrm{der}} \,\} \text{ and } \mathrm{cr}(\mathtt{tl}) = \mathtt{t}.
\end{array} \tag{13.11}$$

One possible definition of well-formedness is to follow the principle that a thread should send only one error message in any fault scenario, which means that any combination of two or more of the three elementary classifications UT^C, UT^S and CrT is considered non-well formed. The well-formed classifications would be

$$UT^C, UT^S, CeT, CrT, UT^C|CeT, UT^S|CeT \text{ and } CeT|CrT.$$

This would be feasible if, e.g., each thread in the system has limited responsibilities and tasks, or an IPC-error is always considered so serious that a shutdown should be performed. This is usually not the case in a multi-threaded control system, though, and we do not formally define well formedness for thread nodes, nor does the current implementation of the fault isolation tool check for it. The extension algorithm for the base graph thread part in Chapter 14 actually only uses the NodeCl function to check for the existence of (unexplained) thread node classification UT^S (see Table 13.4).

Chapter 14

Explaining error messages

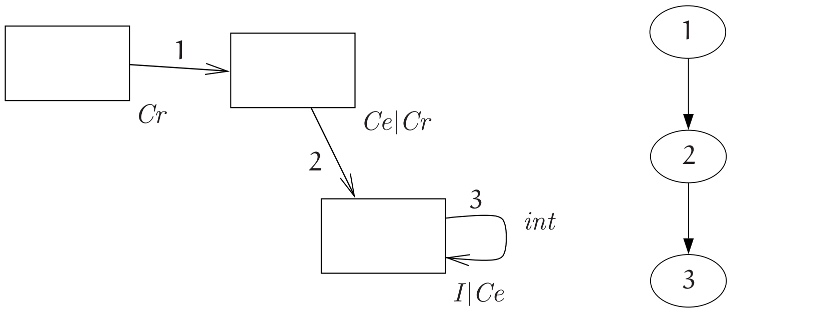
In this chapter, we will show how the multitude of error messages in a fault scenario caused by error condition propagation, described in Chapter 13, can be explained by an automatic procedure.

In the explanation models in Chapter 13, when, e.g., a hardware fault has occured, one error condition in an object triggers an error condition in another, and hence results in one or several "chains" of error messages. This induces a *cause-effect relation* between the error messages in a fault scenario, with the natural, desired, property that it should be a partial order with a unique maximal element. The basic idea in the fault isolation scheme is simply to recreate this relation for a fault scenario, using the error messages and the system model. The maximal elements of the relation are then interpreted as most significant in the fault scenario. With maximal elements in the relation, we mean the elements (error messages) that are not caused (explained) by any other elements, see further Section 14.3.2.

According to the object explanation models in Section 13.1.2, the primary fault in a fault scenario can propagate from object to object in the run-time software system in the form of error conditions, that the objects report to the system log as error messages (Section 13.2.1). An error message can hence be considered caused, or *explained*, by another error message if they involve the same object. For example, in Figure 14.1(a), the first relational error message (denoted $\mathsf{1}$) is explained by the second (denoted $\mathsf{2}$), since they can be assumed to be part of the same error condition propagation chain (see Section 13.1.1). In the same manner, the second relational error message is explained by the internal error message (denoted 3). This cause-effect relation is shown as a graph, called the *explanation graph*, in Figure 14.1(b).

If the complete error condition propagation that has occured in the fault scenario is not represented by the base graph subgraph $< [\mathsf{O}], \mathsf{OL} >$, i.e., if it is not connected, we can attempt to *extend* the graph. In Section 14.2.1 we show how the structural system model from Section 11.2 and Chapter 12 can be used to infer probable error condition chains and extend the subgraph $< [\mathsf{O}], \mathsf{OL} >$.

If the subgraph $< [\mathsf{O}], \mathsf{OL} >$ is not connected, even after extension, the package



(a) Base graph, $<[\mathsf{O}], \mathsf{OL}>$ (b) Explanation graph

Figure 14.1: Example of a connected base graph (object part, $<[\mathsf{O}], \mathsf{OL}>$) and the corresponding cause-effect relation, represented as an explanation graph.

subgraph $<\mathsf{P_O}, \mathsf{PL}>$ may still be connected (see, e.g., Example 13.1). According to the discussion in Section 13.1.2, we can then assume that this graph represents the error condition propagation that has occured, and infer cause-effect relations between error messages also from this graph. We will consider the cause-effect relations between error messages inferred using the package subgraph less reliable than the relations inferred from the object subgraph, though. If the package subgraph $<\mathsf{P_O}, \mathsf{PL}>$ in turn is not connected, it can be extended similar to the object subgraph, as is shown in Section 14.2.2.

According to the discussion in Section 13.1.3, IPC-error messages can both explain, and be explained by, ordinary error messages. Cause-effect relations between IPC-error messages and ordinary error messages can hence be inferred by combining the object part and thread part of the base graph.

As discussed in Section 14.2.3, the thread part of the base graph, subgraph $<\mathsf{T_O}, \mathsf{TL}>$, is only considered a complement to the object part, needed when thread communication problems, as discussed in Section 13.1.3, are part of the fault scenario. In the current implementation, we hence make no attempt to make the thread subgraph connected. However, if the system has sent a relational unknown IPC-error message, we can use the thread part of the system model to infer which thread was the probable complainee, as is further discussed in Section 14.2.3.

It would be desirable to formulate guiding design principles and properties for the control system, so that the property that the cause-effect relation is a partial order with a unique maximal element is guaranteed and also meaningful. This is of course a difficult task to take on in general, and we will only make some brief comments and formulate some general guidelines in Section 14.1.

In Section 14.2, we show how the structural system model from Section 11.2 and Chapter 12 can be used to extend the different parts of the base graph. The complete set of rules used to infer cause-effect relations, or explanations, between error messages using the base graph are given in Section 14.3. We will also show that given that the initial base graph fulfills certain properties, the relation over the error messages is a partial order.

14.1 Assumptions and requirements on the system

A fundamental requirement for the fault isolation scheme to work is of course that the explanation models regarding error condition propagation in the system discussed in Chapter 13 are valid. The error condition propagation must also be traceable, i.e., the propagation must have occured in the software, and sufficient (relational) error messages must have been sent to the log. Fault propagation occuring via hardware dependencies, i.e., in the environment of the system (see Section 10.4), are not recognized in the current approach. In Example 15.4, we demonstrate a fault scenario where fault propagation has occured in the environment.

A guiding design principle for the system and its error messages is that a fault should be reported as close to the actual fault occurrence as possible, where the most relevant information about the fault is available. This is also a requirement for the maximal error messages in the cause-effect relation to be the most significant and suitable to present to a user.

In principle, all faults must have been anticipated by the designers of the system, and corresponding error messages defined, for the maximal error message to be really useful. In practice, though, even in difficult fault situations where the fault isolation scheme does not find a unique explaining error message, or the maximal error message do not give much information, at least many irrelevant messages are sorted out, providing some guidance for a manual fault isolation. See further Section 14.3.2 and Examples 15.4 and 15.2.

For the cause-effect relation to have a *unique* maximal element, that also is significant in the fault scenario, it is of course vital that the single fault assumption is valid for the fault scenario.

The error condition propagation in the system during a fault scenario might encompass loops, i.e., two or more system entities (objects or threads) "blame" each other, forming a loop in one of the base graph subgraphs (see Definition 13.6 and Section 13.4.3). The presence of loops in an initial base graph is undesirable from a system design point of view, since it indicates an unclear internal error handling in the system. Since the base graph is used to form the cause-effect relation, as discussed in Section 14.3, loops in the base graph lead to loops in the cause-effect relation, i.e., such behavior in a fault scenario affect our possibility to find a cause-effect relation on the error messages that is a partial order.

A sensible assumption, that really is a requirement on the system, that will guarantee that the base graph is acyclic also after the extensions described in Section 14.2, is that the initial base graph is acyclic.

Assumption 14.1 (Initial base graph acyclic) *In an initial base graph, i.e., a base graph constructed solely from error messages,*

$$< [O], P_O, T_O, OL, PL, TL >,$$

the directed subgraphs

$$< [O], OL >, < P_O, PL > \ \text{and} \ < T_O, TL >$$

are acyclic (Definition 13.6). If this property holds, the base graph is called acyclic.

Cf. Definition 13.6. Apart from being a general guideline for system designers and implementors for the systems internal error handling, this assumption guarantees, under certain circumstances, that the cause-effect relation on the error messages is a partial order, i.e., do not contain any loops. This will be treated in detail in in Section 14.3.3.

There exist fault scenarios in the ABB Robotics application where the above assumption fails to hold, but they are rare and no such scenario has been studied in this work. We only mention that even if the initial base graph contains loops, the fault isolation scheme still is a useful tool for system designers and implementors to analyze the fault scenario. Also, if the loop(s) in the base graph is (are) not at the root of the error condition propagation, there still exist maximal elements in the cause-effect relation as constructed with the rules in Section 14.3.

We make no use of the timestamp or chronologic order of error messages in the log, other than for clustering purposes. In a system with many concurrent threads and asynchronous communication, the significant messages do not necessarily come first in a fault scenario (see Example 15.1). A common example of such a fault scenario, is if a thread, T_1, is waiting for, e.g., an external device that has suffered a fault, and hence do not respond. Another thread, T_2, then fails when trying to communicate with thread T_1. The error condition propagation following thread T_2's failure then comes before the more significant error message from thread T_1. This situation can of course be avoided by matching the timeouts; thread T_1 should not wait so long, and/or thread T_2 should wait a little longer, but this and like situations are nonetheless common also in well-designed, complex, real-time systems.

14.2 Base graph extension

The initial base graph for a fault scenario, given by the error messages only, is often not connected. That is, the complete error condition propagation is not represented by the error messages. In this section, we will show how the system model from Chapter 12 can be used to *extend* the initial base graph and thus "fill in the gaps". The use of the static system model for this purpose is a kind of "guessing" of dynamic collaborations in the run-time system. The result of such guesswork is of course behefted with some uncertainty. The inferences made by the extension algorithms are analogous to the inferences made by an experienced systems developer when studying the same system log, but the system model and inference methods used by the extension algorithms are of course much cruder.

There are several reasons why the subgraph $< [\mathsf{O}], \mathsf{OL} >$ might not be connected. A common situation, already exemplified in Section 10.3 and Example 13.1, is that the primary fault detection, and the most significant, internal, error message in a fault scenario is produced by a low-level object and a thread separate from the threads where the error condition propagation occurs. Then there often is a gap in the base graph between the relational errors of the error condition propagation chain and the internal error message manifesting the root cause. See, e.g., Example 15.1.

If the fault scenario contains IPC-error messages, the object part of the base graph is often not connected, and it is necessary to combine the thread part and the object part to find the desired cause-effect relation between error messages. This information merge is treated in Section 14.3. See also Example 15.2.

We will show how the object part of the base graph can be extended at two levels of detail. Extension on the object level itself, i.e., for the base graph subgraph $< [\mathsf{O}], \mathsf{OL} >$, is treated in Section 14.2.1. Extension on the package level, i.e., for the base graph subgraph $< \mathsf{P}_\mathsf{O}, \mathsf{PL} >$ is treated in Section 14.2.2.

Our main focus in this fault isolation approach is on the object part of the base graph, i.e., the objects, packages and their corresponding links (complaints), but sometimes the thread part is necessary to get the complete picture. The thread extension algorithm introduced Section 14.2.3 makes no attempt to make the thread part of the base graph, i.e., the subgraph $< \mathsf{T}_\mathsf{O}, \mathsf{TL} >$, connected, and is only meant to explain the relational unknown IPC-error messages.

14.2.1 Class Extend Graph Algorithm

In this section, we will show how the system model can be used to make the base graph subgraph $< [\mathsf{O}], \mathsf{OL} >$ connected, or at least more connected than before. To this end, we identify the objects in $< [\mathsf{O}], \mathsf{OL} >$ where it seems to be a gap in the observed error condition propagation chain. That is, we identify the objects that have complaints on them, but have no-one to blame and have not sent a chain rooting error message, i.e., an internal (*int*) error message or an IPC-error message.

To this end, the well-formed object node classifications $\mathsf{WF}_{\mathsf{C}_\mathsf{O}}$ from Equation (13.5) in Section 13.4.4 are partitioned into two sets:

$$\begin{aligned} \mathsf{NC}_{\mathsf{C}_\mathsf{O}} &= \{\, U,\, U|Ce,\, Ce \,\} \\ \mathsf{OK}_{\mathsf{C}_\mathsf{O}} &= \{\, I, I|Ce,\, Cr,\, U|Cr,\, U|Ce|Cr,\, Ce|Cr, \\ &\qquad UT,\, Ce|UT,\, CrT,\, Ce|CrT \,\} \end{aligned} \tag{14.1}$$

Intuitively, objects with classification from $\mathsf{NC}_{\mathsf{C}_\mathsf{O}}$, read as “No Complainee” or “Needs Complainee”, have complaints on them, but no good idea of who to blame. The three object node classifications in $\mathsf{NC}_{\mathsf{C}_\mathsf{O}}$ are visualized in the leftmost column of Figure 13.8.

The server chain function

We use the system model to search for possible complainees by forming chains of associated classes. A *server chain* is a sequence of associated classes in the system model,

$$[c_0, \ldots, c_n] \in \mathsf{SC}$$

where $c_1, \ldots, c_n \in \mathsf{Classes}$ and $n \geq 1$. SC is the set of all server chains. We also define the selector function

$$\text{last}([c_0, \ldots, c_n]) = c_n,$$

with the natural extension to sets of server chains.

The exact definition we propose for a server chain, is given by the *server chain function*

$$\text{sc} : \mathsf{Classes} \times \mathbb{N} \to \mathsf{SC},$$

which intuitively represents a search for possible complainees for an object in the system model ($\mathbb{N} = \{1, 2, \ldots\}$ are the natural numbers). The server chain function uses the class part of the system model, $< \mathsf{Classes}, \mathsf{A}, \twoheadrightarrow >$ as defined in Chapter 12.

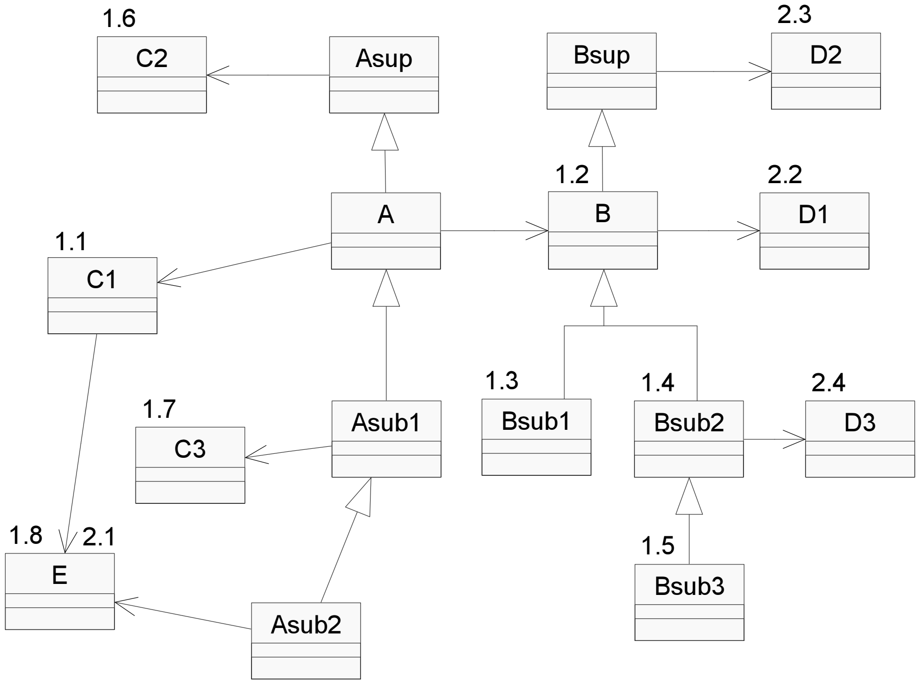
As mentioned in Section 12.1, the class associations in the system model may be stereotyped ($\mathsf{S_A}$). In the current version of the class extension algorithm, the only functionality is that certain associations can be ignored when searching for complainees. In the ABB Robotics application, the association stereotypes whose associations are ignored by sc are

$$\mathsf{TS} := \{<<\text{ Enqueue }>>, <<\text{ Subscribe }>>\}.$$

As described in Section 11.2.1, these stereotypes model that the association is implemented with an asynchronous thread communication scheme, and any problem with that collaboration will be reported with an IPC-error message. The definition of sc is recursive, and is shown in Table 14.1. The server chain function is designed to explore all associations in the system model through which a class may be dependent on another class. In Algorithm 14.1, the output from the server chain function is matched against generalized objects in the base graph, i.e., instantiated objects that have manifested themselves in the fault scenario.

The class for which the search in the system model starts, is called the *original search class.* The sc-definition is illustrated in Figure 14.2, where the result of our actual implementation is shown. The numbering by the classes in Figure 14.2(a) and in the leftmost column of the table in Figure 14.2(b) give the order in which our implementation finds the server chains. The left number is the search depth and the right enumerates the chains (<search depth>.<nr>).

It is assumed that the original search class may be a superclass of the instantiated class that has manifested itself in the fault scenario. Hence the function $\text{sc}(c, 0)$ in Table 14.1 includes all subclasses of the original search class. In $\underline{\text{sc}}(c, 1)$,



(a) Part of the $< \mathsf{Classes}, \mathsf{A}, \rightarrowtriangle >$ graph from a possible system model. The annotations at the top of the classes give the order in which the server chains for the original search class A are found (<search depth>.<nr>) by our implementation of sc.

<dpth>.<nr>	Server chain	Member of
1.1	A - C1	$\underline{\mathrm{sc}}(\mathrm{A}, 1)$
1.2	A - B	
1.3	A - Bsub1	$\mathrm{sc}_{\mathrm{d}}(\mathrm{A}, 1)$
1.4	A - Bsub2	
1.5	A - Bsub3	
1.6	A - C2	$\overline{\mathrm{sc}}(\mathrm{A}, 1)$
1.7	Asub1 - C3	$\underline{\mathrm{sc}}(\mathrm{A}, 1)$
1.8	Asub2 - E	
2.1	A - C1 - E	$\underline{\mathrm{sc}}(\mathrm{A}, 2)$
2.2	A - B - D1	
2.3	A - B - D2	$\overline{\mathrm{sc}}(\mathrm{A}, 2)$
2.4	A - Bsub2 - D3	$\underline{\mathrm{sc}}(\mathrm{A}, 2)$

(b) The possible complainees for the original search class A, in the order given by our implementation of the server chain function sc.

Figure 14.2: Example of the server chain function sc defined in Table 14.1.

$$
\begin{array}{lcl}
\mathrm{sc}(c,0) & = & \{\, [c_0] \mid c_0 \twoheadrightarrow c \text{ or } c_0 = c \,\} \\
\underline{\mathrm{sc}}(c,1) & = & \{\, [c_0,c_1] \mid c_0 \in \mathrm{sc}(c,0) \text{ and } c_0 \xrightarrow{s} c_1 \in \mathsf{A}, s \notin \mathsf{TS} \,\} \\
\overline{\mathrm{sc}}(c,1) & = & \{\, [c_0,c_1] \mid c_0 = c, c \twoheadrightarrow \tilde{c} \text{ and } \tilde{c} \xrightarrow{s} c_1 \in \mathsf{A}, s \notin \mathsf{TS} \,\} \\
\mathrm{sc}_d(c,1) & = & \{\, [c_0,c_1] \mid [c_0,\tilde{c}] \in \overline{\mathrm{sc}}(c,1) \cup \underline{\mathrm{sc}}(c,1) \text{ and } c_1 \twoheadrightarrow \tilde{c} \,\} \\
\mathrm{sc}(c,1) & = & \underline{\mathrm{sc}}(c,1) \cup \overline{\mathrm{sc}}(c,1) \cup \mathrm{sc}_d(c,1)
\end{array}
$$

For $k > 1$:

$$
\begin{array}{lcl}
\underline{\mathrm{sc}}(c,k) & = & \{\, [c_0,\ldots,c_k] \mid [c_0,\ldots,c_{k-1}] \in \mathrm{sc}(c,k-1) \\
 & & \quad \text{and } c_{k-1} \xrightarrow{s} c_k \in \mathsf{A}, s \notin \mathsf{TS} \,\} \\
\overline{\mathrm{sc}}(c,k) & = & \{\, [c_0,\ldots,c_k] \mid [c_0,\ldots,c_{k-1}] \in \overline{\mathrm{sc}}(c,k-1) \cup \underline{\mathrm{sc}}(c,k-1), \\
 & & \quad c_{k-1} \twoheadrightarrow \tilde{c} \text{ and } \tilde{c} \xrightarrow{s} c_k \in \mathsf{A}, s \notin \mathsf{TS} \,\} \\
\mathrm{sc}_d(c,k) & = & \{\, [c_0,\ldots,c_k] \mid [c_0,\ldots,c_{k-1},\tilde{c}] \in \overline{\mathrm{sc}}(c,k) \cup \underline{\mathrm{sc}}(c,k) \text{ and } c_k \twoheadrightarrow \tilde{c} \,\} \\
\mathrm{sc}(c,k) & = & \underline{\mathrm{sc}}(c,k) \cup \overline{\mathrm{sc}}(c,k) \cup \mathrm{sc}_d(c,k)
\end{array}
$$

Table 14.1: The server chain function definition.

all associations, i.e., servers, of these classes are enumerated. $\overline{\mathrm{sc}}(c,1)$ explores the associations of the superclasses of the original search class. We follow the general rule that class information should be kept as specific as possible, hence the server chains returned by $\overline{\mathrm{sc}}(c,1)$ have the original search class as first member, even though the association was found in a superclass.

The function $\mathrm{sc}_d(c,1)$ adds a server chain for all subclasses of the found servers, so that the extension algorithm also matches these against generalized objects in the base graph. As can be seen in Figure 14.2, our implementation do not calculate the parts of sc as defined in Table 14.1 in order, but the server chains given by sc_d are enumerated immediately after the corresponding chain given by $\underline{\mathrm{sc}}$ or $\overline{\mathrm{sc}}$.

Following the rule mentioned above, that class information should be as specific as possible, superclasses of the servers found by $\underline{\mathrm{sc}}(c,1)$ and $\overline{\mathrm{sc}}(c,1)$ are not returned. The extension algorithm (Algorithm 14.1 below) explicitly has to search for generalized objects that are superclasses of the returned servers.

The enumeration of servers then continues with server chains involving more than one association, i.e., $k > 1$ in Table 14.1. The definition of $\mathrm{sc}(c,k)$ for $k > 1$ is a direct generalization of $\mathrm{sc}(c,1)$.

The extension algorithm

To define the class extension algorithm, we need a graph theory function that indicates whether two objects in $< [\mathsf{O}], \mathsf{OL} >$ belong to the same connected component (Definition 13.5):

$$\text{connected} : [\mathsf{O}] \times [\mathsf{O}] \longrightarrow \mathbb{B}.$$

Simple algorithms for this purpose are well-described in the literature, see, e.g., [73]. Our proposed class extension algorithm is given in Algorithm 14.1.

Algorithm 14.1 ClassExtendGraph

The algorithm searches the system model for possible complainees for base graph object nodes that needs it, and extends the base graph accordingly.

Syntax: ClassExtendGraph($< \mathsf{O}, \mathsf{OL} >$, $< \mathsf{Classes}, \multimap\!\!\triangleright, \mathsf{A} >$)

Input: Base graph, System model

Output: Extended base graph ($< \mathsf{O}, \mathsf{OL} >$)

The algorithm is in two parts, a main algorithm that loops through all object nodes in the base graph, and one subroutine, classExtendGraph, that does the actual extending. The pseudo-code for the main procedure is given in Figure 14.3(a), and the pseudo-code for the subroutine is given in Figure 14.3(b).

□

The algorithm searches the system model for possible complainees (lines 2 and 3 in the subroutine, Figure 14.3(b)) for all generalized objects in O that need a complainee (lines 1 and 4 in the main procedure) according to Equation (14.1).

Only well formed objects in $[\mathsf{O}]$ are considered, line 2. A non-well formed object is interpreted as a design fault, or that the message log cluster, discussed in Section 13.3, includes error messages from several fault scenarios.

The matching of the servers returned by the server chain function sc against generalized objects in the base graph is done in two steps, for reasons discussed when the server chain function was defined above. First a direct match is checked for, line 4, and then the algorithm looks in the base graph for superclasses of the found server, lines 9-10. The last part of the conditions on lines 4 and 10 prohibits the algorithm from adding loops to the base graph.

If a complainee candidate already present in O is found, (lines 4 and 10 in Figure 14.3(b)) the corresponding extra generalized objects and object links are added to O and OL. We have here chosen to use only the first possible complainee found by the server chain function. Other alternatives are to add *all* possible complainees up to a certain search depth, or to use a priority scheme, as will be used for the package extend algorithm in Section 14.2.2. The reason for this choice is simplicity, and that the system model $< \mathsf{Classes}, \mathsf{A}, \multimap\!\!\triangleright >$ used in the ABB Robotics application is specific enough for it to work. The extension performed may then of course depend on the order in which the server chain function enumerates the possible complainees. This approach has shown to be suitable for the ABB Robotics application, though, since there mostly exists only one suitable complainee in the system model.

As already mentioned, when adding generalized objects to the base graph, we also need to impose the base graph property that for $\mathsf{o}, \mathsf{o}' \in \mathsf{O}$ with class(o) $\multimap\!\!\triangleright$

```
for the most specific representative of each equivalence class [o] ∈ O do
  if cl(o) ∉ WF_{C_o} then
    issue warning;
  elsif cl(o) ∈ NC_{C_o} then
    classExtendGraph(o);
  end if
end for
```

(a) The main procedure.

```
classExtendGraph(o ∈ O)
for k = 1 : max_depth do
  for s ∈ sc(class(o), k) do
    if ∃o′ ∈ O, class(o′) = last(s) and not connected([o], [o′]) then
      add s to base graph;
      comment All added links are given classification rel_der
      return
    end if
    if s ∉ sc_d(class(o), k) then
      if ∃o′ ∈ O, class(o′) ∈ asc(last(s)) and not connected([o], [o′]) then
        add s to base graph;
        comment All added links are given classification rel_der
        return
      end if
    end if
  end for
end for
```

(b) The subroutine classExtendGraph.

Figure 14.3: The Class Extend Graph algorithm, Algorithm 14.1.

class(o'), generalized objects corresponding to the classes between class(o) and class(o') in the inheritance relation partial order, are present in the base graph.

The constant max_depth on line 2 in Figure 14.3(b), is a design parameter for the extension algorithm. It is used to chose how much we need and can rely on the static class information in the system model when extending the graph. In the ABB Robotics application, the "density" of the error messages in the system is such that we have found max_depth= 2 to be a reasonable choice.

Note also that `classExtendGraph`, if successful, adds a Cr to the classification of the argument object, and hence moves it from $\mathsf{NC}_{\mathsf{C_O}}$ to $\mathsf{OK}_{\mathsf{C_O}}$ (Equation (14.1)). The classification of the found complainee object gets an added Ce, which does not change the classification group. All added objects have classification $Ce|Cr$, and hence belong in $\mathsf{OK}_{\mathsf{C_O}}$.

14.2.2 Package Extend Graph Algorithm

In this section, we show how the system model from Chapter 12, specifically $<\mathsf{P_C}, \mathsf{PD}>$, can be used to make the package subgraph of the base graph, $<\mathsf{P_O}, \mathsf{PL}>$, connected. As for the class extend graph algorithm in the previous section, we identify the packages that has complaints on them, but no good idea of who to blame.

To this end, the well formed package classifications $\mathsf{WF}_{\mathsf{C_P}}$ from Equation (13.8) are partitioned into two parts:

$$\begin{array}{rcl} \mathsf{NC}_{\mathsf{C_P}} & = & \{\,U,\, U|Ce,\, Ce,\, \mathsf{0}\,\} \\ \mathsf{OK}_{\mathsf{C_P}} & = & \{\,I, I|Ce,\, Cr,\, U|Cr,\, U|Ce|Cr,\, Ce|Cr, \\ & & \;T,\, Ce|T,\, U|T,\, U|Ce|T,\, U|I,\, U|I|Ce\,\} \end{array} \tag{14.2}$$

where packages with classification from $\mathsf{NC}_{\mathsf{C_P}}$ needs a complainee.

As for the class extension algorithm, we need a graph theory function that indicates whether two packages in the directed graph $<\mathsf{P_O}, \mathsf{PL}>$ belong to the same connected component (Definition 13.5):

$$\text{connected} : \mathsf{P_O} \times \mathsf{P_O} \longrightarrow \mathbb{B}.$$

Simple algorithms for this purpose are well-described in the literature, see, e.g., [73]. Our proposed package extension algorithm is given in Algorithm 14.2.

Algorithm 14.2 PackageExtendGraph

The algorithm searches the system model for possible complainees for packages that needs it, and extends the base graph accordingly.

Syntax: PackageExtendGraph($<\mathsf{P_O}, \mathsf{PL}>$, $<\mathsf{P_C}, \mathsf{PD}>$)

Input: Base graph, System model

Output: Extended base graph ($<\mathsf{P_O}, \mathsf{PL}>$)

Pseudo code for the algorithm is given in Figure 14.4. □

```
1  for p_o ∈ P_O do
2     if cl(p_o) ∉ WF_{C_P} then
3        issue warning;
4     elsif cl(p_o) ∈ NC_{C_P} then
5        P̃_O = {p̃_o ∈ P_O : pmap(p_o) --→ pmap(p̃_o) ∈ PD}
6        if for any p̃_o ∈ P̃_O, cl(p̃_o) ∈ OK_{C_P} and (pmap(p̃_o) non-global) and
          not connected(p_o, p̃_o) then
7           Add p_o --pack--> p̃_o to PL;
8        elsif for any p̃_o ∈ P̃_O, cl(p̃_o) ∈ OK_{C_P} and (pmap(p̃_o) global) and
          not connected(p_o, p̃_o) then
9           Add p_o --pack--> p̃_o to PL;
10       elsif for any p̃_o ∈ P̃_O, cl(p̃_o) ∈ NC_{C_P} and (pmap(p̃_o) non-global) and
          not connected(p_o, p̃_o) then
11          Add p_o --pack--> p̃_o to PL;
12       elsif for any p̃_o ∈ P̃_O, cl(p̃_o) ∈ NC_{C_P} and (pmap(p̃_o) global) and
          not connected(p_o, p̃_o) then
13          Add p_o --pack--> p̃_o to PL;
14       end if
15    end if
16 end for
```

Figure 14.4: The Package Extend Graph algorithm, Algorithm 14.2.

The algorithm loops through all packages in $\mathsf{P_O}$. For non-well formed packages, a warning is given and that package is ignored (line 2).

For each base graph package $\mathrm{p_o}$ with classification from the set $\mathsf{NC_{C_P}}$, Equation (14.2), we find all other packages that have manifested themselves in the fault scenario *and* are related to $\mathrm{p_o}$ through the package dependency relation PD in the system model (the set $\tilde{\mathsf{P}}_\mathsf{O}$ at line 5).

The package part of the system model, $< \mathsf{P_C}, \mathsf{PD} >$, is of course coarser than the pure class part. The search depth for the package extension algorithm, i.e., the maximal number of consecutive package links added to the base graph, has hence been chosen to 1, instead of 2 as for the class extension algorithm, Algorithm 14.1. Since there also is no inheritance for packages, the search in the system model is much simplified when compared to the class extension algorithm.

One possibility would be to add a package link to $< \mathsf{P_O}, \mathsf{PL} >$ for *all* the packages in $\tilde{\mathsf{P}}_\mathsf{O}$. We have instead chosen to add only one package link, according to a priority (see the lines 6, 8, 10 and 12). The last part of the conditions, **not** connected$(\mathrm{p_o}, \tilde{\mathrm{p}}_\mathrm{o})$, prohibits the algorithm from adding package links that could create loops in the base graph.

The first choice of package link (or explanation) to add to the base graph, leads to a package that already has someone to blame, i.e., belongs to the set $\mathsf{OK_{C_P}}$ in

Equation (14.2) (lines 6 and 8). Recall from Section 11.2.3 and Section 12.1 that packages declared global in the system model are interpreted as having a package dependency to them from every other package in the system model. To explore the package dependencies specifically added to the system model first, non-global packages are preferred to global packages.

If no complainee candidate package with classification from $\mathsf{OK}_{\mathsf{C_P}}$ was found, we turn to packages that themselves need a complainee, i.e., with classification from the set $\mathsf{NC}_{\mathsf{C_P}}$. As before, non-global packages are preferred to global ones (lines 10 and 12). By using this priority scheme, the exact extension performed may depend not only on how the possible complainees in the system model are ordered (line 5), but also in which order the base graph package nodes are enumerated (line 1). The extension strategy, both for Algorithm 14.2 and Algorithm 14.1, is of course a design decision for the fault isolation scheme, and the chosen approach has shown suitable for the ABB Robotics application.

When a package satisfying the conditions is found, a corresponding package link is added (lines 7, 9, 11 and 13). Note that the link classification for the added package link is *pack*, as opposed to *obj*-package links, that are induced by object links (see Section 13.4.3). See Example 15.3 for a real fault scenario where the class extend algorithm failed, and the package extend algorithm was needed.

14.2.3 Thread Extend Graph Algorithm

The thread extend graph algorithm is only needed when IPC-error messages with unknown complainee are part of the fault scenario. The algorithm attempts to explain relational unknown IPC-error messages using the thread dependencies in the thread part of the system model ($< \mathsf{T_C}, \mathsf{TD} >$).

For the ABB Robotics application, we have actually only found it necessary to consider the thread collaborations according to the Enqueue-pattern (see Section 11.2), when the provider thread (the client in the system model) do not provide the consumer thread (the server) with enough data. For modularity reasons, the consumer does not know the provider, and the resulting IPC-error condition is hence relational unknown, with direction client.

In the current implementation of the fault isolation tool, we hence only try to extend the thread part of the base graph to explain relational unknown IPC-error messages with direction client. That is, a thread $\mathsf{t} \in \mathsf{T_O}$ is considered in need of a complainee thread only if

$$UT^C \in \text{NodeCl}(\mathsf{t}) \text{ and } CrT \notin \text{NodeCl}(\mathsf{t}), \tag{14.3}$$

where the symbols UT^C and CrT are defined in Table 13.4 and Equation (13.11) in Section 13.4.4. Furthermore, the found thread dependency in the system model must have the relevant stereotype, in our case <<Enqueue>>.

The proposed thread extension algorithm is given in Algorithm 14.3.

Algorithm 14.3 ThreadExtendGraph

The algorithm searches the system model for possible complainees for base graph thread nodes that needs it, and extends the base graph accordingly.

Syntax: ThreadExtendGraph($< \mathsf{T_O}, \mathsf{TL} >$, $< \mathsf{T_C}, \mathsf{TD} >$)

Input: Base graph, System model

Output: Extended base graph ($< \mathsf{T_O}, \mathsf{TL} >$)

Pseudo code for the algorithm is given in Figure 14.5. □

1 **for** all $\mathtt{t_o} \in \mathsf{T_O}$ **do**
2   **if** $UT^C \in \mathrm{NodeCl}(\mathtt{t_o})$ and $CrT \notin \mathrm{NodeCl}(\mathtt{t_o})$ **then**
3     $\mathsf{T_{dep}} = \{\mathtt{td} \in \mathsf{TD} \mid \mathrm{server}(\mathtt{td}) = \mathrm{tmap}(\mathtt{t_o})$ and $\mathrm{stereotype}(\mathtt{td}) = <<\textit{Enqueue}>>\}$
4   **for** $\mathtt{td} \in \mathsf{T_{dep}}$ **do**
5   **if** $\exists \tilde{\mathtt{t}}_\mathtt{o} \in \mathsf{T_O}$ s.t. $\mathrm{tmap}(\tilde{\mathtt{t}}_\mathtt{o}) = \mathrm{client}(\mathtt{td})$ **then**
6   add $\mathtt{tl} = \mathtt{t_o} \overset{relT_{der},-}{\rightarrow} \tilde{\mathtt{t}}_\mathtt{o}$ to TL
7   **end if**
8   **end for**
9   **if** $\mathrm{client}(\mathsf{T_{dep}}) \cap \mathrm{tmap}(\mathsf{T_O})$ is empty **then**
10   **for** $\mathtt{td} \in \mathsf{T_{dep}}$ **do**
11   add a thread node $\tilde{\mathtt{t}}_\mathtt{o}$ with $\mathrm{tmap}(\tilde{\mathtt{t}}_\mathtt{o}) = \mathrm{client}(\mathtt{td})$ to $\mathsf{T_O}$
12   add $\mathtt{tl} = \mathtt{t_o} \overset{relT_{der},-}{\rightarrow} \tilde{\mathtt{t}}_\mathtt{o}$ to TL
13   **end for**
14   **end if**
15   **end if**
16 **end for**

Figure 14.5: The Thread Extend Graph algorithm, Algorithm 14.3.

The algorithm loops through all thread nodes $\mathsf{T_O}$ in the base graph and picks out the thread nodes fulfilling the condition in Equation (14.3) (lines 1 and 2). For each such thread node, we find all possible complainees in the system model ($\mathsf{T_{dep}}$ at line 3).

The extension is then attempted in two steps; first the possible complainees from the system model ($\mathsf{T_{dep}}$) are matched against threads that have already manifested themselves in the fault scenario. If such a thread node is found, a thread link is added (lines 4–8).

The thread link added at line 6 (and 12) is given the classification $relT_{der}$, i.e., a derived relational thread link (see Section 13.4.3). Since the added thread link has not been sent as an error message by an object in the system, the sending object is left undefined, denoted with the symbol "–".

The second attempt at extending for a thread node fulfilling the conditions in Equation (14.3), is given by the lines 9–15. Since a thread with problems often is silent, see Section 13.1.3, the conjunction between the possible complainees in the system model and the threads in the base graph (line 9) often is empty . If T_{dep} is not empty (line 10), we assume that the complainee thread is silent, and add it to the base graph "manually", together with the corresponding thread link (lines 11 and 12).

In the thread extension algorithm all suitable dependencies are added, see lines 4 and 10, and not just one. The reason we have chosen this approach, is that there are much fewer thread dependencies than associations in the system model, and the stereotype requirement limits the number even more. In the system model for the ABB system, there are actually no threads with more than one thread dependency stereotyped << *Enqueue* >>. An example with the thread extension algorithm is given in Example 15.3.

14.3 The cause-effect relation

In this section, we will specify how we construct the cause-effect relation, or *explanation relation*, over the error messages using the, possibly extended, base graph representing a fault scenario. A visualization of the explanation relation is called an *explanation graph*.

14.3.1 A relation over the error messages

The cause-effect relation will be defined over the set of links

$$\mathsf{L} = \mathsf{OL} \cup \mathsf{TL} \cup \mathsf{TL}_e$$

where TL and OL, defined in Section 13.4.3, are all object and thread links in the base graph, i.e., all error messages in the system log cluster, plus the derived links added by the class and thread extension algorithms in Section 14.2. The here introduced link set TL_e is initially empty, and will be used to add extra, derived, thread links to L, for reasons further explained below.

We define the cause-effect, or explanation relation, R, as a set of *explanations*

$$l_1 \xrightarrow{k} l_2 \in \mathsf{R} \subseteq \mathsf{L} \times \mathsf{K} \times \mathsf{L}.$$

As mentioned in the introduction to this chapter, certain explanations inferred from the base graph are considered less reliable than others. We formalize this by the *trust level*,

$$\mathsf{K} = \{1, 2\},$$

where explanations with level 2 are stronger than explanations with level 1. The weights on the explanations are only to get a reliability estimate, so we will allow ourselves to call R a relation over L, i.e., over the error messages. The projection

$$\{ (\mathsf{l}_1, \mathsf{l}_2) \mid \exists \mathsf{k} \text{ s.t. } \mathsf{l}_1 \xrightarrow{\mathsf{k}} \mathsf{l}_2 \in \mathsf{R} \}$$

is the actual (binary) cause-effect relation over the error messages. The intuitive interpretation of an element $\mathsf{l}_1 \xrightarrow{\mathsf{k}} \mathsf{l}_2 \in \mathsf{R}$, is that the error message (link) represented by l_1 is explained, with trust level k, by the link represented by l_2.

Construction of R and the explanation graph. The explanation relation is constructed with a set of *explanation rules*, corresponding to the explanation models discussed in Section 13.1. Each rule applies to a local situation in a base graph, and adds an explanation to R for every occurrence of that situation. The final result hence depends only on the base graph and not on the order in which the rules are applied. In the explanation rules, we assume that the base graph is well formed (see Section 13.4.4). R is the smallest set fulfilling all the explanation rules, and if two explanations are identical but have different trust levels, the weaker explanation is removed.

To define the explanation rules, we consider error condition propagation according to the two explanation models given in Section 13.1 (object and thread, respectively). The explanation rules are further divided into four groups, according to how the error condition propagation represented by the base graph is explained:

- Pure object level explanations
- Explanations interior in package
- Package to package explanations
- IPC error message explanations

These groups and their respective explanation rules are presented in detail below.

To enhance readability, the complete explanation relation is not visualized as an explanation graph in the implemented tool (*DrRobot*). Derived links in L that are non-maximal in R are not shown, i.e., we project the relation on the real error messages when visualizing the final explanation graph. If a derived link is maximal, it is kept in order not to discard important information, though (see Example 15.2 and 15.3). We also remove redundant information in the form of transitive edges (explanations), that are implicit in the graph visualization. If the transitive explanation is stronger than the minimal strength for the explanations along the longer path, the transitive explanation is kept, though (see Example 15.1, Figure 15.2).

Notation used in this section We introduce some standard notation for use in this section. In what follows, $\mathsf{ol}, \mathsf{ol}_1$ and ol_2 are object links where $\mathsf{ol}_1 \neq \mathsf{ol}_2$;

$\mathtt{tl}, \mathtt{tl}_1$ and $\mathtt{tl}_2$ are thread links where $\mathtt{tl}_1 \neq \mathtt{tl}_2$; $\mathtt{l} \in \mathsf{L}$ is an arbitrary link, and $\mathtt{p}$ is a package. In the figures, we also use superscripts, e.g., $\mathtt{ol}_1^1, \mathtt{ol}_1^2$, to denote different links. We define the two auxiliary functions

$$\text{nCe}\colon \mathsf{OL} \to [\mathsf{O}]$$
$$\text{sender}\colon \mathsf{L} \to [\mathsf{O}]$$

as

$$\text{nCe}(\mathtt{ol}) \quad = \quad \begin{cases} [\text{ce}(\mathtt{ol})] & \text{if } \text{cl}(\mathtt{ol}) \in \{\, \mathit{rel}, \mathit{rel}_{\text{der}} \,\} \\ [\text{cr}(\mathtt{ol})] & \text{if } \text{cl}(\mathtt{ol}) = \mathit{rel}_u \\ \texttt{undef} & \text{if } \text{cl}(\mathtt{ol}) = \mathit{int} \end{cases}$$
$$\text{sender}(\mathtt{l}) \quad = \quad \begin{cases} [\text{cr}(\mathtt{l})] & \text{if } \mathtt{l} \in \mathsf{OL} \\ [\text{obj}(\mathtt{l})] & \text{if } \mathtt{l} \in \mathsf{TL} \text{ and } \text{cl}(\mathtt{l}) \neq \mathit{relT}_{\text{der}} \\ \texttt{undef} & \text{if } \mathtt{l} \in \mathsf{TL} \text{ and } \text{cl}(\mathtt{l}) = \mathit{relT}_{\text{der}}. \end{cases}$$

For an object link, the function nCe returns the object that "needs a complainee", whereas the function sender returns the object that sent the error message. The special symbol `undef` has the properties that it is not equal to anything nor is a member of any set.

Object level explanations

When an object $\mathtt{o}_1$ complains on another object $\mathtt{o}_2$ which in turn has made a further complaint, we assume that an object-to-object error condition propagation, as described in Section 13.1, has occured from $\mathtt{o}_2$ to $\mathtt{o}_1$. Hence the first error message is explained by the second one. Rule 1a explains object-to-object fault propagation using the $< [\mathsf{O}], \mathsf{OL} >$ part of the base graph, and Rule 1b recognizes that an IPC-error may start an object-to-object fault propagation. This is a strong form of explanation.

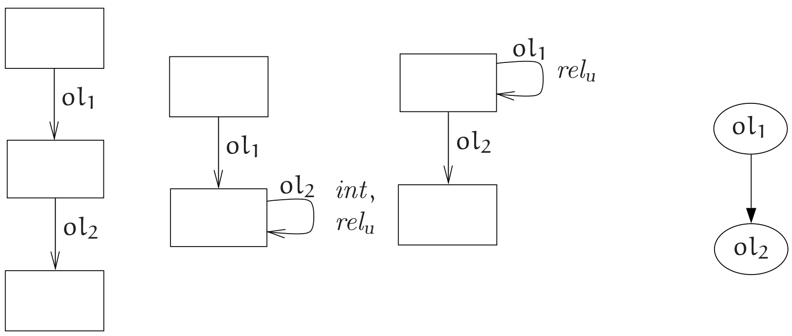
Rule 1a: $\text{nCe}(\mathtt{ol}_1) = [\text{cr}(\mathtt{ol}_2)]$ and $[\text{cl}(\mathtt{ol}_1), \text{cl}(\mathtt{ol}_2)] \neq [\mathit{rel}_u, \mathit{rel}_u] \Longrightarrow$
$\mathtt{ol}_1 \xrightarrow{2} \mathtt{ol}_2 \in \mathsf{R}$

Rule 1b: $\text{nCe}(\mathtt{ol}) = [\text{obj}(\mathtt{tl})] \Longrightarrow \mathtt{ol} \xrightarrow{2} \mathtt{tl} \in \mathsf{R}$

The base graph situations covered by these rules are exemplified in Figure 14.6 and 14.7. In Figure 14.7(a), the symbol relT_*^* represents any of the possible IPC-error message classifications relT, relT_u^C, relT_u^S and $\mathit{relT}_{\text{der}}$.

Interior package explanations

Sometimes the occured fault propagation is not represented in the graph $< [\mathsf{O}], \mathsf{OL} >$, even after the class extend graph algorithm in Section 14.2.1 has been applied. We then fall back on the assumption that error conditions easily propagate within a package, as objects from the same package often cooperate closely. Note that the explanation relation needs not be exactly "correct", in that it recreates exactly



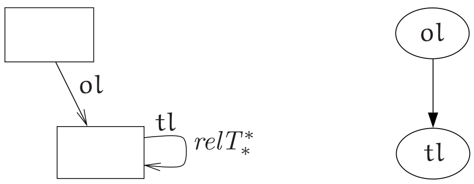
(a) Three base graph situations. (b) The explanation graph

Figure 14.6: Three base graph situations ($< [\mathsf{O}], \mathsf{OL} >$) covered by Rule 1a. The corresponding explanation graphs are the same for all three situations.

what caused what in the system during the fault scenario. It is enough if it has the correct maximal elements, and makes educated guesses about which error messages are connected by an error condition propagation. The explanations derived from the package membership are considered weak explanations.

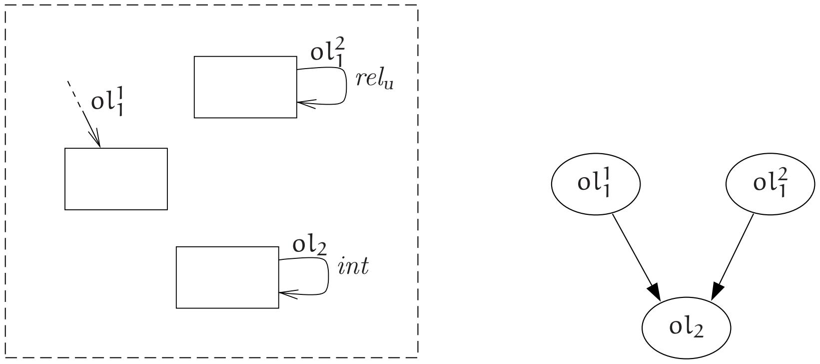
We let an *int* error message respectively an IPC-error message weakly explain other object links in the same package (Rule 2a respectively Rule 2b below). An *int* error message and an IPC-error message in the same package would not be well formed.

Rule 2a: $\mathrm{nCe}(\mathsf{ol}_1) \in \mathsf{p}, \mathrm{cr}(\mathsf{ol}_2) \in \mathsf{p}$ and $\mathrm{cl}(\mathsf{ol}_2) = int \Longrightarrow \mathsf{ol}_1 \xrightarrow{1} \mathsf{ol}_2 \in \mathsf{R}$



(a) The base graph. (b) The explanation graph

Figure 14.7: The base graph situations ($< [\mathsf{O}], \mathsf{OL} >$) covered by Rule 1b.



(a) The base graph (b) The explanation graph

Figure 14.8: Examples of base graph situations ($< \mathsf{O}, \mathsf{P_O}, \mathsf{OL} >$) covered by Rule 2a.

Rule 2b: $\text{nCe}(\mathtt{ol}) \in \mathtt{p}, \text{obj}(\mathtt{tl}) \in \mathtt{p} \Longrightarrow \mathtt{ol} \xrightarrow{1} \mathtt{tl} \in \mathsf{R}$

The base graph situations that these rules cover are exemplified in Figure 14.8 and 14.9.

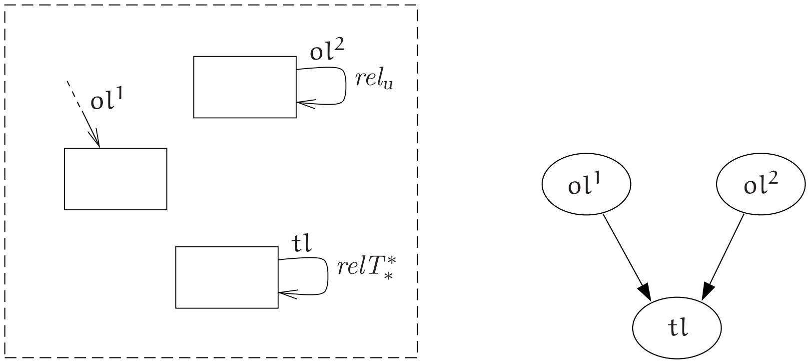
Package to package explanation

There might also be object links in need of explaining in a package, but no *int* or IPC-error message present, so Rule 2a or 2b cannot be employed. If there is a package link in PL from the package to another, we make the assumption that the error conditions have propagated between the packages, and let the object links in the first package be explained by the error messages in the second. These explanations are also of the weaker kind.

Rule 3a: $\text{nCe}(\mathtt{ol}_1) \in \mathtt{p}_1, \text{cr}(\mathtt{ol}_2) \in \mathtt{p}_1, \text{ce}(\mathtt{ol}_2) \in \mathtt{p}_2$ and $\mathtt{p}_1 \neq \mathtt{p}_2 \Longrightarrow$
$\mathtt{ol}_1 \xrightarrow{1} \mathtt{ol}_2 \in \mathsf{R}$

Rule 3b: $\text{nCe}(\mathtt{ol}) \in \mathtt{p}_1, \mathtt{p}_1 \xrightarrow{pack} \mathtt{p}_2 \in \mathsf{PL}$ and $\text{sender}(\mathtt{l}) \in \mathtt{p}_2 \Longrightarrow$
$\mathtt{ol} \xrightarrow{1} \mathtt{l} \in \mathsf{R}$

In Rule 3a, the object link $\mathtt{ol}_2$ induces a package link $\mathtt{pl} = \mathtt{p}_1 \xrightarrow{obj} \mathtt{p}_2 \in \mathsf{PL}$. The base graph situations that these rules cover are exemplified in Figure 14.10 and 14.11. In Figure 14.11(a), * is used to symbolize any of the object or thread link classifications *int*, rel_u and $relT^*_*$. The non-well formed cases (see Section 13.4.4) for the second package must of course be omitted.



(a) The base graph (b) The explanation graph

Figure 14.9: Examples of base graph situations ($<\mathsf{O}, \mathsf{P_O}, \mathsf{OL}>$) covered by Rule 2b.

IPC error message explanation

The final set of rules concerns the explanation of IPC-error messages, i.e., error messages sent due to thread problems.

We consider a relational thread link weakly explained by errors in the complainee thread (Rule 4a). If there is one relational unknown IPC error message and a derived relational thread link ($relT_{\mathrm{der}}$) from the same thread, added by the thread extension algorithm described in Section 14.2.3, the derived thread link strongly explains the other (Rule 4b). Finally, a chain of thread messages indicates causality (Rule 4c).

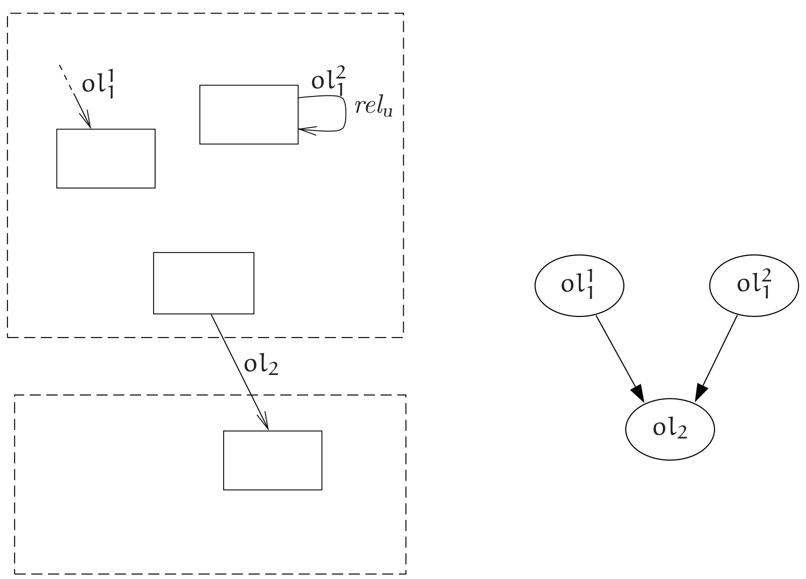
Rule 4a: $\mathrm{cl}(\mathtt{tl}) \in \{\, relT, relT_{\mathrm{der}} \,\}$ and $\mathrm{th}(\mathtt{ol}) = \mathrm{ce}(\mathtt{tl}) \Longrightarrow \mathtt{tl} \xrightarrow{1} \mathtt{ol} \in \mathsf{R}$

Rule 4b: $\mathrm{cl}(\mathtt{tl}_1) \in \{relT^C_u, relT^S_u\}, \mathrm{cl}(\mathtt{tl}_2) = relT_{\mathrm{der}}$ and $\mathrm{cr}(\mathtt{tl}_1) = \mathrm{cr}(\mathtt{tl}_2) \Longrightarrow$
$\mathtt{tl}_1 \xrightarrow{2} \mathtt{tl}_2 \in \mathsf{R}$

Rule 4c: $\mathrm{cl}(\mathtt{tl}_1) \in \{\, relT, relT_{\mathrm{der}} \,\}$ and $\mathrm{ce}(\mathtt{tl}_1) = \mathrm{cr}(\mathtt{tl}_2) \Longrightarrow \mathtt{tl}_1 \xrightarrow{2} \mathtt{tl}_2 \in \mathsf{R}$

The base graph situations covered by these rules are exemplified in Figure 14.12, 14.13 and 14.14. For Rule 4c, Figure 14.14(a), it should be noted that $\mathtt{tl}_2$ can also be a relational unknown IPC-error message.

As pointed out in Section 13.1.3, a thread with problems often is silent in the fault scenario. For that reason, we include a more "active" rule among the explanation rules, that adds an extra element to $\mathsf{TL_e}$, and hence L, if a silent thread has more than one complaint on it from other threads. If a silent thread has exactly one complaint on it, that corresponding link will serve as the maximal element in the explanation graph (see, e.g., Example 15.3).



(a) The base graph

(b) The explanation graph

Figure 14.10: Examples of base graph situations ($< \mathsf{O}, \mathsf{P_O}, \mathsf{OL}, \mathsf{PL} >$) covered by Rule 3a.

To formulate the rule, we formally call a thread $\mathsf{t} \in \mathsf{T_O}$ *silent* if

$$\{\,\mathsf{tl} \in \mathsf{TL} \mid \mathrm{cr}(\mathsf{tl}) = \mathsf{t}\,\} = \emptyset \text{ and } \{\,\mathsf{ol} \in \mathsf{OL} \mid \mathrm{th}(\mathsf{ol}) = \mathsf{t}\,\} = \emptyset.$$

We also introduce the function complaints : $\mathsf{T_O} \longrightarrow 2^{\mathsf{TL}}$

$$\mathrm{complaints}(\mathsf{t}) = \{\,\mathsf{tl} \in \mathsf{TL} \mid \mathrm{ce}(\mathsf{tl}) = \mathsf{t}\,\}.$$

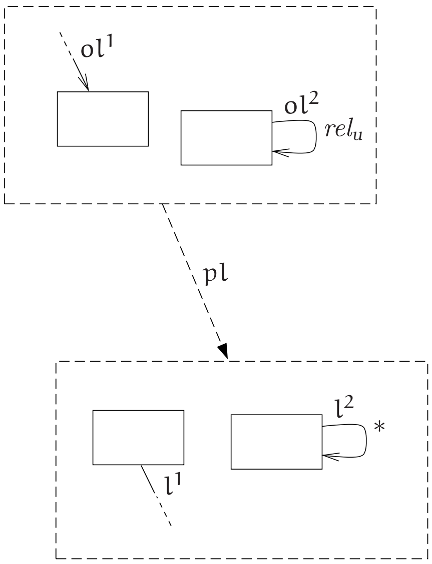
The rule then becomes

Rule 5: $\mathsf{t} \in \mathsf{T_O}$ silent, and $|\mathrm{complaints}(\mathsf{t})| > 1 \Longrightarrow$
add $\mathsf{tl}_e = \mathsf{t} \overset{intT_{der},-}{\rightarrow} \mathsf{t}$ to TL_e, and $\mathsf{tl} \overset{1}{\rightarrow} \mathsf{tl}_e \in \mathsf{R}\ \forall \mathsf{tl} \in \mathrm{complaints}(\mathsf{t})$

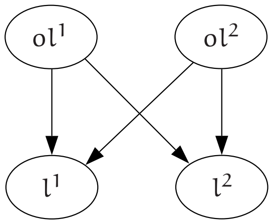
See Example 15.2. The link classification $intT_{\mathrm{der}}$ was introduced in Equation (13.2), Section 13.4.3.

14.3.2 Interpretation of the explanation relation

The explanation relation, base graph and the fault isolation scheme as a whole has at least two different usages:



(a) The base graph



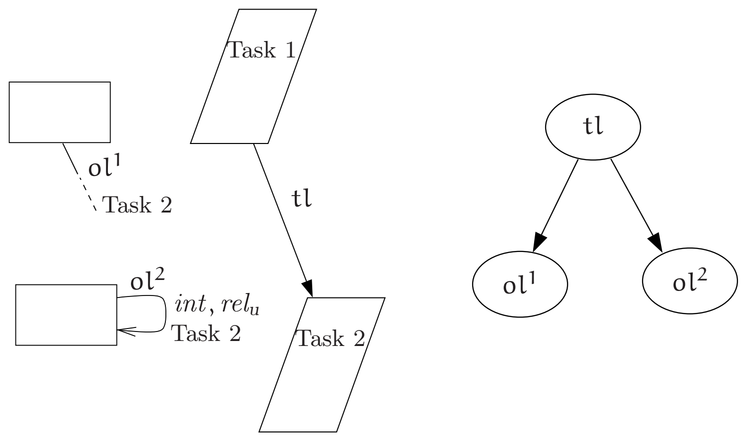
(b) The explanation graph

Figure 14.11: Examples of base graph situations ($< \mathsf{O}, \mathsf{P_O}, \mathsf{OL}, \mathsf{PL} >$) covered by Rule 3b.

- To be a part of the automatic fault isolation layer discussed in Chapter 10.
- To serve as a tool for system designers and developers to analyze fault scenarios encountered, e.g., during system integration tests or on the field.

To serve as an analysis tool in the development process, we need a nice visualization of the base graph and the explanation graph, as have already been presented in Section 13.4.1 and Section 14.3 so far.

For the fault isolation scheme to serve as part of a fault handling layer, we need as output for a given fault scenario the maximal error messages, and preferably an estimate of the reliability, i.e., how confident we are in that the fault isolation scheme has reached a sensible conclusion. With a *maximal error message*, or *maximal element*, we mean an element of L that is not explained, in R, by any other element of L. If there is a unique maximal element for R, then it of course explains all elements in L. Note that the maximal element need not be an error message, but can be a derived thread link, pinpointing a specific thread. The explanation relation needs not be exactly "correct", in that it recreates exactly what caused what in the system during the fault scenario. It is enough if R has the correct maximal elements, and makes an educated guess about how the error condition propagation took place.



(a) The base graph

(b) The explanation graph

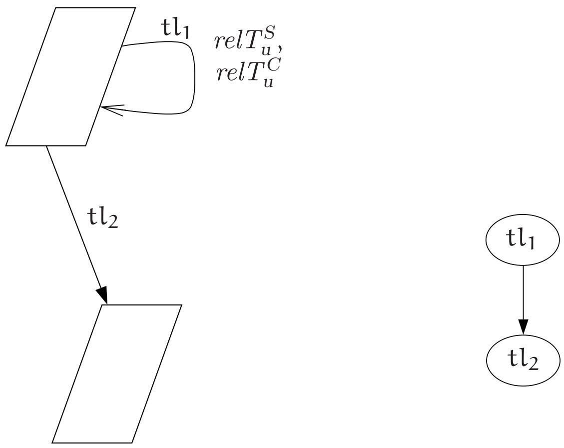
Figure 14.12: Examples of base graph situations ($< \mathsf{O}, \mathsf{OL}, \mathsf{T_O}, \mathsf{TL} >$) covered by Rule 4a.

The main question when considering the reliability of the fault isolation scheme is, how well do the base graph, and hence the explanation graph, recreate the actual fault scenario? Given the arguments put forth in this thesis so far, the answer would be that the fault scenario is well recreated, given that the assumptions discussed in Section 14.1 are fulfilled. An estimate of the reliability is then in principle obtained by estimating how well the assumptions are fulfilled in a specific fault scenario.

This is of course a difficult task, which has not been extensively studied in this work, yet. We here propose to address the problem by checking certain graph theoretic properties of the base graph and explanation graph, that are directly dependent on the properties of the system and fault scenario. The interesting graph properties, for ordinary directed graphs, are:

- Connected (Definition 13.5)
- Unique maximal element
- Acyclic (Definition 13.6)

Preferably, the explanation graph, that is an ordinary directed graph, has all these properties. We can note that if there are no IPC-error messages, the explanation graph is connected iff $< \mathsf{P_O}, \mathsf{PL} >$ is connected. If in addition $< \mathsf{O}, \mathsf{OL} >$ is connected, then all explanations will be of the strong kind, and it also indicates that the error condition propagation in the system is traceable, i.e., that enough error messages have been sent to the log.



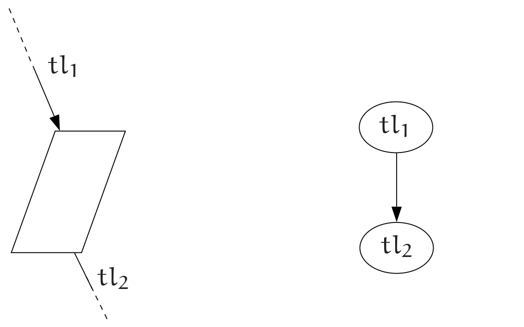
(a) The base graph (b) The explanation graph

Figure 14.13: Examples of base graph situations ($< \mathsf{T_O}, \mathsf{TL} >$) covered by Rule 4b.

Multiple maximal elements in R can have a number of causes (note that multiple maximal elements in a directed graph can be caused by the graph not being connected).

- Multiple faults have occured, or bad clustering of log.
- The error condition propagation is not traceable.
- No error message directly corresponds to the occured fault, or fault propagation in the environment (see Example 15.4).

As already mentioned in Section 14.1, loops in the base graph subgraphs indicate an unclear internal error handling in the system. Recall from Section 14.1, Assumption 14.1, that a base graph is called acyclic if all three subgraphs are acyclic. Note that the extension algorithms presented in this chapter do not introduce cycles, so if the initial base graph is acyclic (Assumption 14.1), then the extended base graph is also acyclic. Unfortunately, the base graph acyclic do not formally guarantee that the explanation graph is acyclic, due to the perpendicular nature of the thread part and object part of the base graph (we have yet to encounter examples of this, though). In Section 14.3.3 below, we will present sufficient properties that guarantee that R is acyclic. They can be taken as a design guideline for the control system.



(a) The base graph (b) The explanation graph

Figure 14.14: Examples of base graph situations ($< \mathsf{T_O}, \mathsf{TL} >$) covered by Rule 4c.

14.3.3 When is the explanation graph acyclic

We will show that the relation R can be made to a partial order (i.e., transitive, reflexive and anti-symmetric) under certain circumstances. To that end, we introduce R's transitive and reflexive closure, $\bar{\mathsf{R}}$.

$$
\begin{array}{lll}
l_1 \xrightarrow{k} l_2 \in \mathsf{R} & \Rightarrow & l_1 \xrightarrow{k} l_2 \in \bar{\mathsf{R}} \\
l_1 \xrightarrow{k_1} l' \in \mathsf{R} \text{ and } l' \xrightarrow{k_2} l_2 \in \mathsf{R} & \Rightarrow & l_1 \xrightarrow{min(k_1,k_2)} l_2 \in \mathsf{R} \\
l \in \mathsf{L} & \Rightarrow & l \xrightarrow{2} l \in \mathsf{R}
\end{array}
$$

The transitive closure is taken by stating that if there is an explanation path from message l_1 to message l_2, we say that there is a direct explanation between them, with a strength of the minimum strength along the path. The formulation is recursive. If we also can assure that $\bar{\mathsf{R}}$ is anti-symmetric, i.e., acyclic, it would be a partial order on the error messages.

Even if we assume that the base graph is acyclic, Assumption 14.1, the perpendicular nature of the object part respectively the thread part of the base graph constitute a problem when attempting to formally prove that R is anti-symmetric. A sufficient condition is that that there are no IPC-errors in the base graph, i.e., if the thread part is essentially empty. We will comment further on this after stating the theorem and proof.

Theorem 14.1 R partial order
If the base graph is well-formed and the base graph subgraphs $< [\mathsf{O}], \mathsf{OL} >$ and $< \mathsf{P_O}, \mathsf{PL} >$ are acyclic, and TL is empty, then $\bar{\mathsf{R}}$ is anti-symmetric and hence a partial order. □

Trivially, $\bar{\mathsf{R}}$ is anti-symmetric iff R is acyclic. The proof is mostly a matter of pointing out some facts.

Proof (Theorem 14.1) The distinction between a strong and weak explanation, respectively, is of no importance here and will be ignored.

The conditions in Rules 1–5 defining R are all in terms of the base graph, and the effects of the rules are all on R. Also, no element is present in R that is not supported by at least one rule. Hence we can regard the construction of R as a sequential and exhaustive application of the rules, where the final result does not depend on the order of application.

Since TL is empty, the applicable rules are 1a, 2a, 3a and 3b. We will show that R is acyclic, which is equivalent with $\bar{\mathsf{R}}$ anti-symmetric.

We will prove the claim by exhaustively applying the rules to construct an originally empty relation $\tilde{\mathsf{R}}$. Since no thread links are present in the base graph, the applicable rules are 1a, 2a, 3a and 3b.

Rule 1a: Each application of the rule uses a specific object in $[\mathsf{O}]$ to add an edge between two object links in OL to $\tilde{\mathsf{R}}$. Explanations are added from a relational link with the object as complainee, to all links with the object as complainer. An explanation is also added from relational unknown links sent by the object to relational links with the object as complainer. Note that $\mathrm{cl}(\mathsf{ol}_1) \neq \mathit{int}$ due to the definition of nCe. The rule trivially does not introduce loops in $\tilde{\mathsf{R}}$ for the links involved in the same object.

Since the rule for a specific object in $[\mathsf{O}]$ only adds edges to $\tilde{\mathsf{R}}$ from incoming links to non-incoming links, and no two objects in the directed graph $< [\mathsf{O}], \mathsf{OL} >$ are connected, no cycles in $\tilde{\mathsf{R}}$ are introduced by Rule 1a (it can be noted that the above argument would cover also Rule 1b).

Rule 2a: For each package $\mathsf{p} \in \mathsf{P_O}$ that contains an object, $\mathsf{o} \in [\mathsf{O}]$, that has sent an *int* error message, the rule introduces an edge in $\tilde{\mathsf{R}}$ from all relational links with the complainee in the package and all relational unknown in the package, to the *int* link. Neither Rule 1a nor 2a introduce any edge in $\tilde{\mathsf{R}}$ from an internal link (*int*), so no cycles can be introduced (it can be noted that the above argument would cover also Rule 2b). Note that a relational link with either o as complainer or with the complainee in another package, would not be well formed (cf. Figure 14.8).

Rule 3a and 3b: Both rules introduce edges in $\tilde{\mathsf{R}}$ according to the base graph subgraph $< \mathsf{P_O}, \mathsf{PL} >$. Rule 3b introduces explanations in $\tilde{\mathsf{R}}$ from certain object links in a package $\mathsf{p}_1 \in \mathsf{P_O}$ that is complainer of a package link $\mathsf{p}_1 \overset{pack}{\rightarrow} \mathsf{p}_2 \in \mathsf{PL}$, to all links sent by objects in p_2. The rule does not introduce cycles by itself since $< \mathsf{P_O}, \mathsf{PL} >$ is acyclic. The only way for cycles to be introduced in the hitherto $\tilde{\mathsf{R}}$, is through Rule 1a, if the combination of object links OL and package links PL have formed a cycle in the base graph. Due to the definition of *obj* links in PL, any such cycle would be present also in $< \mathsf{P_O}, \mathsf{PL} >$, though.

Rule 3a introduces explanations in $\tilde{\mathsf{R}}$ from certain object links in a package $\mathtt{p}_1 \in \mathsf{P_O}$ that is complainer of a package link $\mathtt{p}_1 \stackrel{obj}{\rightarrow} \mathtt{p}_2 \in \mathsf{PL}$, to the object link(s) that has (have) induced the package link. The "certain object links" in package $\mathtt{p}_1$ are relational unknown object links sent by objects in $\mathtt{p}_1$ or relational object links with their complainee in $\mathtt{p}_1$. An *int* (or $relT_*^*$) in $\mathtt{p}_1$ would not be well-formed. The rule do not introduce cycles in $\tilde{\mathsf{R}}$ by itself or in combination with Rule 3b and Rule 1a, since $< \mathsf{P_O}, \mathsf{PL} >$ is acyclic.

Since the above rules are the only applicable under the given conditions, we have $\tilde{\mathsf{R}} = \mathsf{R}$, and the claim is proved. ■

The reason we must exclude IPC-error messages for the theorem, is that the base graph object part, $< [\mathsf{O}], \mathsf{OL} >$ and $< \mathsf{P_O}, \mathsf{PL} >$, might overlap in a unfortunate way with the thread part, $< \mathsf{T_O}, \mathsf{TL} >$. The rules 1b, 2b and 3b introduce explanations from an object link to a thread link assuming the error condition propagation started with the IPC-error, and Rule 4a introduces explanations from a thread link to an object link, assuming that the error condition causing the object error message is connected to the problem that made the thread unable to fulfill its responsibilities toward the rest of the system. If Rule 4a is removed, we can easily extend the above proof and show that $\bar{\mathsf{R}}$ is a partial order under Assumption 14.1.

In Example 15.2 and Example 15.3, real fault scenarios containing IPC-error messages are presented, where the corresponding (final) explanation graphs are connected, have a unique maximal element and are acyclic.

Chapter 15

Implementation and Examples

The fault isolation scheme presented here is implemented and tested on a large scale industrial control system developed by ABB Robotics.

15.1 *DrRobot*

The fault isolation tool, with the working name *DrRobot*, is a prototype implementation of the fault isolation scheme presented in this thesis. The tool hence strives to recreate a fault scenario from the system log and a structural system model.

The core of *DrRobot*, i.e., the construction, manipulation and internal representation of the base graph and explanation graph, consists, by a low estimate, of 3000 lines of C++ code[1]. The complete tool, including visualization and user interaction, runs under Windows NT™.

DrRobot reads clustered system logs as ordinary text files, that are produced directly by the control system. Any number of logs, also multiple copies of the same log, can be manipulated by the tool at the same time. The signature for each error message, as defined in Section 13.2.1, is currently supplied by a lookup-table in the *DrRobot* source code. The error message signatures needed to handle the fault scenarios from the ABB Robotics industrial control system presented in this thesis, were acquired by studying the actual control system source code and from interviews with the system developers at ABB Robotics. It can be noted that the signatures fit very naturally in the object oriented architecture, and are easy to determine even for someone otherwise unfamiliar with the implementation. The lookup-table is of course easily extendable and exchangeable. Error message signatures can also be supplied directly in the log.

The system model used in the ABB application was created with the UML tool Rational Rose™, see Section 11.2. *DrRobot* accesses the system model using the

[1]See [38] for a thorough discussion on how to count lines of code.

Automation interface, Rose Extensibility Interface, supplied with Rational Rose™. Automation, formerly known as OLE-Automation, is an inter-application communication interface supplied by Windows NT™. For use in a production system, the relevant parts of the system model probably need to be exported from Rational Rose™ and represented more compactly and with less overhead. See Chapter 12 for suitable data structures.

To create the internal representation of the initial base graph for a specific fault scenario, see Figure 13.6 in Section 13.4.2, *DrRobot* uses the error message signatures, and adds inheritance and package information from the system model.

The two views of the base graph presented in Section 13.4.1 are visualized. The explanation graph is constructed from the current base graph as described in Section 14.3 and also visualized. See Section 15.2 for several examples. The user applies the extension algorithms presented in Section 14.2 to the current base graph via the menu-driven interface. The nodes in the different graphs are automatically placed in a fairly intelligent manner, but the nodes can also be dragged with the mouse to increase readability.

In the base graph views, derived links, and also derived nodes, are shown in gray to distinguish them from nodes and links stemming from actual error messages. All derived links in the base graph are currently given the ID number `0`. As discussed in Section 14.3, these links are not shown in the explanation graph if they are not maximal. A thread link added to the explanation graph by Rule 5 in Section 14.3.1 is in the current implementation given the ID number −42, to distinguish it from the derived links added to the base graph (see Example 15.2). The different trust levels of the explanations in the explanation graph are visualized by making the weaker explanation arrows gray.

15.2 Examples

The ABB Robotics industrial robot control system was presented in Section 11.3, and consists of approximately 2 million lines of code and ca. 500 classes. The used system model, which do not cover the whole system, has ca. 150 classes and 20 logical packages. Due to space considerations, not to clutter the presentation and not to disclose company internal information, the system model will not be shown. The error messages signatures for the studied fault scenarios were acquired directly from the source code of the control system, and through interviews of the developers at ABB Robotics.

The scheme has been tested on a number of real fault scenarios (10–15), several of which are quite similar on our level of abstraction, though. Examples were also found where the internal fault handling in the core system needs to be improved. Where the system assumptions in Section 14.1 were fulfilled, the scheme was able to pinpoint the most significant error message, or the dubious thread, if silent, in all cases where the fault propagation had occured in the software and not in the environment.

We will here present three examples of fault scenarios with distinctly different characteristics, where the fault isolation scheme was successful, and one example where fault propagation has occured in the environment of the system, and no unique maximal error message exists. The fault scenarios are handled with the implemented fault isolation tool, *DrRobot*, that also has produced the figures.

Example 15.1 Field bus fault scenario, cont.
The example is a continuation of Example 13.1 in Section 13.4.1, where the ABB robot control system is running a user-defined program, when a field-bus unit suffers a fault.

The Figure 15.1 shows the initial base graph, already shown in Figure 13.3, and the corresponding explanation graph constructed with the rules in Section 14.3.1. Already initially, using only the local information in the error message signatures, it is possible to form a connected explanation relation over the error messages, using the package explanation model and the corresponding rules in Section 14.3.1. The explanation from error message 71139 to 71061 is added by Rule 3a and the explanation from 71061 to 71107 is added by Rule 2a.

By applying Algorithm 14.1 (ClassExtendGraph) the base graph, suitable associations are found in the system model and derived links and an in-between generalized object are added to $< \mathsf{O}, \mathsf{OL} >$, as shown in Figure 15.2. The conclusion before and after the extension is basically the same, the error message 71107 is closest to the actual fault, but Algorithm 14.1 strengthens the result somewhat. After the extension, $< \mathsf{O}, \mathsf{OL} >$ is connected and the package explanation model is not needed to make the explanation graph connected. The explanation from 71139 to 71061 according to Rule 3a is still added, though. The explanation from 71061 to 71107 according to Rule 2a is also still added, but is outweighed by the stronger explanation added by Rule 1a. Note how the non-maximal derived links are removed from the explanation graph. □

Example 15.2 IPC fault scenario, cont.
The example is a continuation of Example 13.2 in Section 13.4.1. Here we will use the visualizations of the different graphs produced by the fault isolation tool, instead of the edited versions used in Example 13.2.

We repeat what happened from in the fault scenario from Example 13.2. The thread called `cab task` got hung up in a communication with an external device, and two other threads, `eio task` and `safevt task`, failed after trying to contact `cab task`. After clustering and filtering of operational messages in the log, only three error messages remained. The clustered system log for the fault scenario is shown in Figure 15.3.

The initial base graph and explanation graph are shown in Figure 15.4. The thread `cab task` is silent in the fault scenario, and only shows up in the base graph because it is the complainee of relational IPC-error messages. In the explanation graph, Figure 15.4(c), the explanation from 90002 to 90001 is added by Rule 2b. The two other explanations, and the extra link, a derived internal thread link for the thread `cab task` (in the *DrRobot* generated figure denoted with -42), are added by Rule 5. Note that all explanations are of the weaker kind.

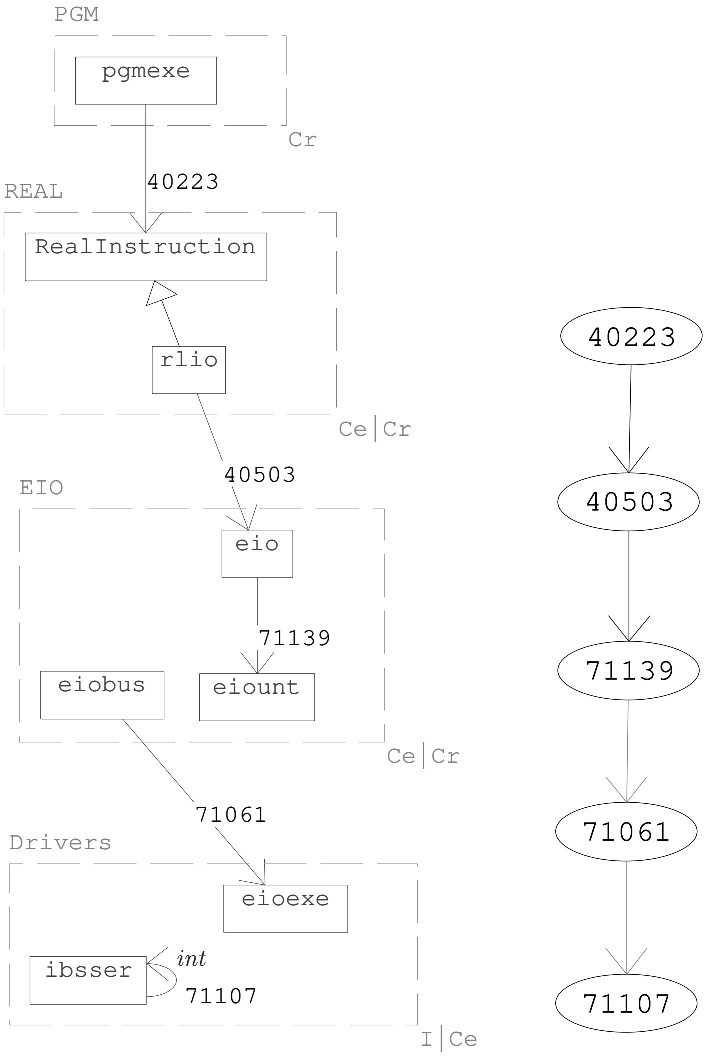


Figure 15.1: The initial base graph ($< \mathsf{O}, \mathsf{P_O}, \mathsf{OL}, \mathsf{PL} >$) and explanation graph from the fault scenario in Example 15.1.

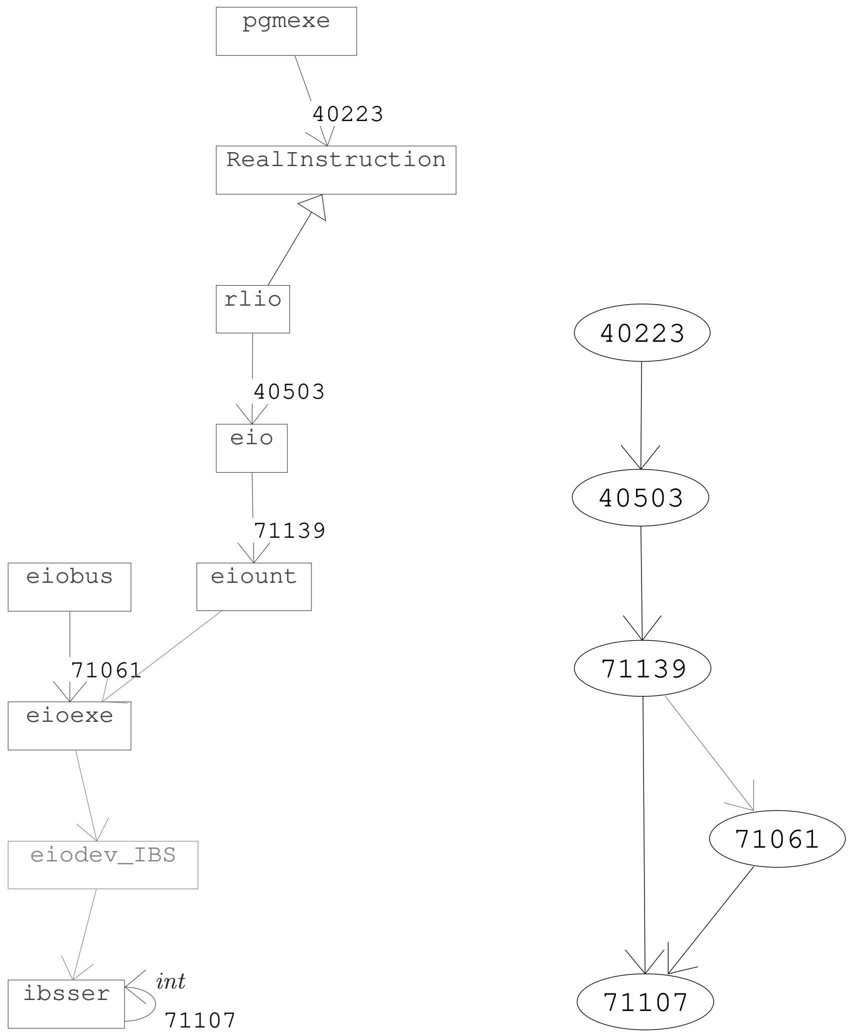


Figure 15.2: The extended base graph ($<\mathsf{O},\mathsf{OL}>$) and explanation graph from the fault scenario in Example 15.1.

```
27.  71156 IPC queue full 0102 03:36.26
Description\Reason:
- The ipc queue was full, when sending to trap routine.
  errno = 3d0002

1. 90001 safdev.c 211 IPC distribute error -9 0102 03:37.4
safdev.c 211 IPC distribute error -9
2. 90002 safevtts.c 678 internal error response 0102 03:37.4
safevtts.c 678 internal error response
```

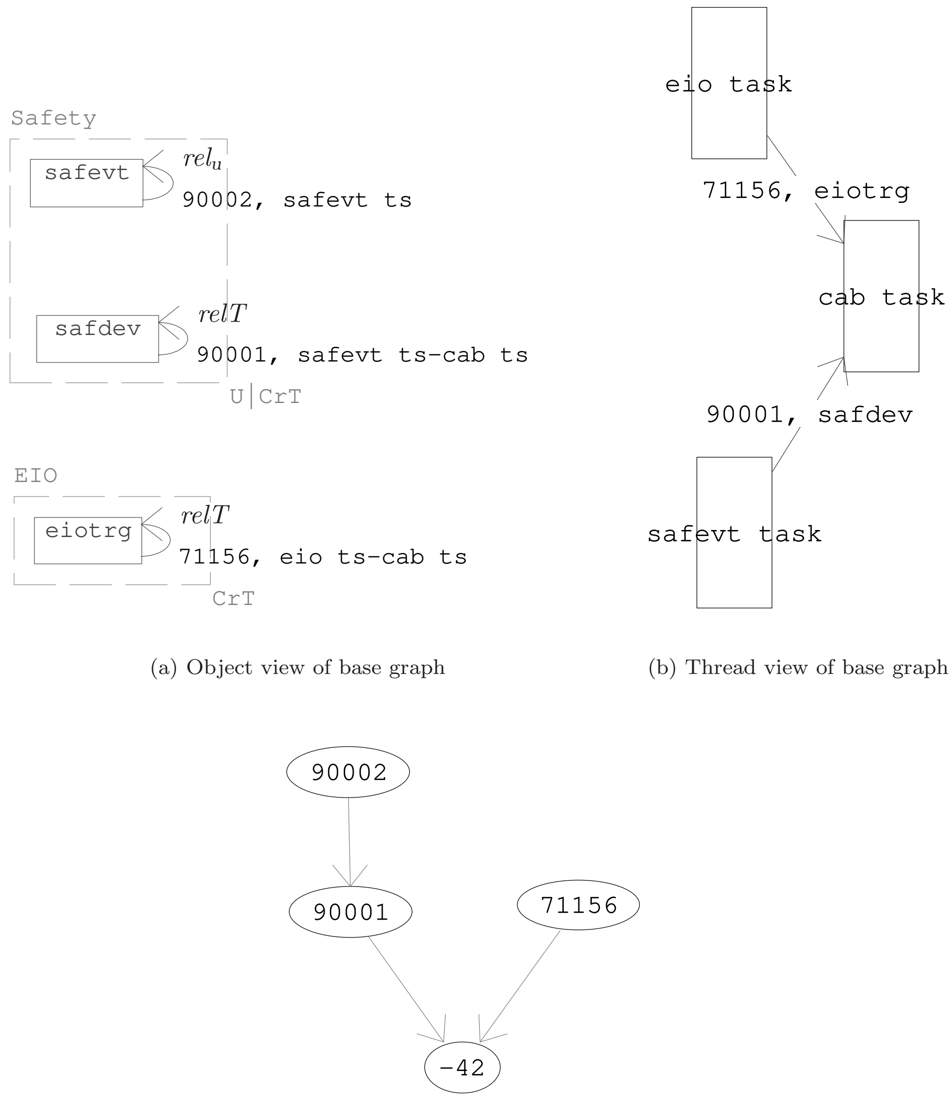
Figure 15.3: The clustered system log for the fault scenario in Example 15.2.

The explanation graph is already connected and has a unique maximal element, hence we can consider the fault scenario sufficiently recreated. If we apply Algorithm 14.1 (ClassExtendGraph) to the initial base graph anyway, the only difference is that a relational derived object link (rel_{der}) is added from the base graph object `safevt` to `safdev` in Figure 15.4(a), hence the explanation from 90002 to 90001 in the explanation graph, Figure 15.4(c) is "upgraded" to a strong explanation. The Algorithm 14.2 (PackageExtendGraph) has no effect. □

Example 15.3 Real time fault scenario, cont.
The example is a continuation of Example 13.3 in Section 13.4.1. The scenario is that during normal operation of the system, one thread feeds another with data according to the enqueue-pattern (see Section 11.2.1).

The fault is that the data-providing thread, the client in the system model, cannot provide data for some reason, e.g., because of a too high computational load. The result is a fault scenario where the server thread in the enqueue-pattern, `servo task` in the base graph, fails due to data starvation and sends a relational unknown IPC-error message with direction backward (client). After clustering the log, two error messages remain, one relational unknown and the IPC-error message. The initial base graph and explanation graph is shown in Figure 15.5. As seen in Figure 15.5, the initial base graph do not contain enough information to recreate the fault scenario especially well. By applying Algorithm 14.2 (PackageExtendGraph) and Algorithm 14.3 (ThreadExtendGraph), we get the extended base graph and corresponding explanation graph shown in Figure 15.6. The Algorithm 14.1 (ClassExtendGraph) has no effect.

The derived thread link in Figure 15.6(b) corresponds to a thread dependency in the system model with stereotype << *Enqueue* >>, and the thread dependency has the opposite direction as the derived thread link. Note that the derived thread link, by *DrRobot* given id-number 0, is maximal in the explanation graph, and hence not removed. □

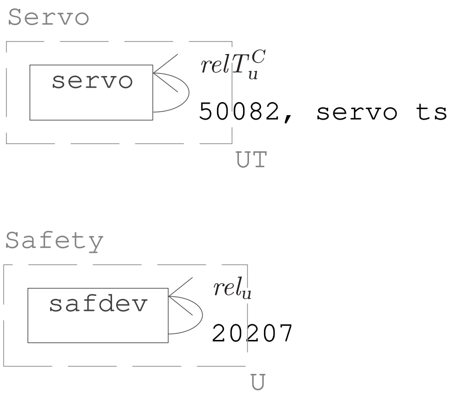


(a) Object view of base graph

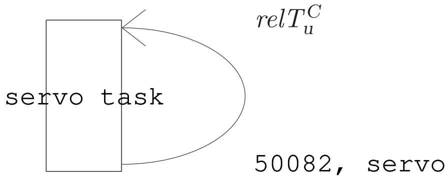
(b) Thread view of base graph

(c) Explanation graph

Figure 15.4: The initial base graph and explanation graph from the fault scenario in Example 15.2.



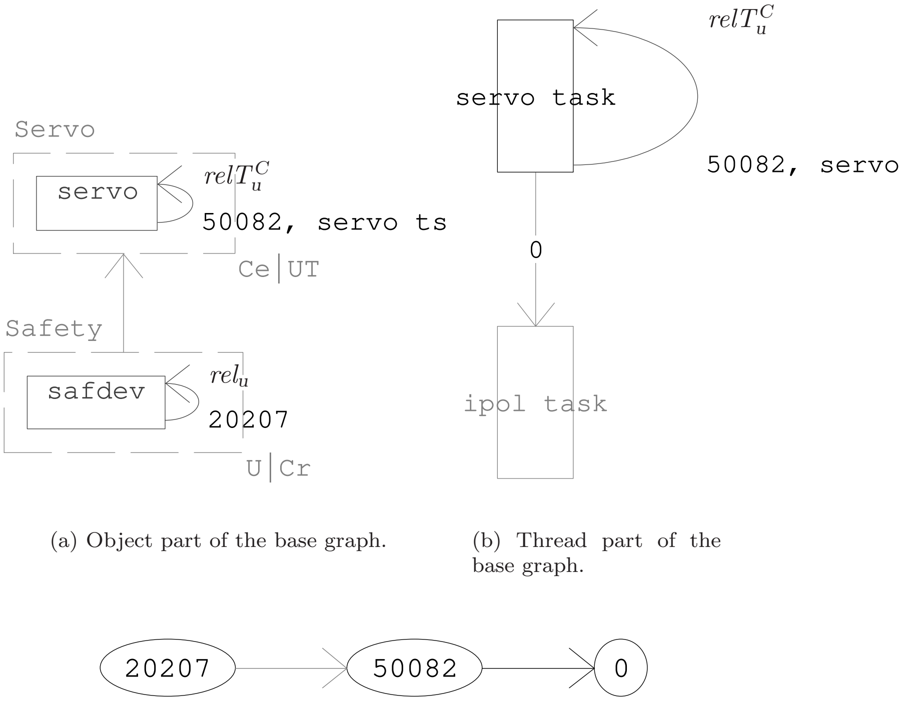
(a) Object part of the base graph.



(b) Thread part of the base graph.

(c) The explanation graph.

Figure 15.5: The initial base graph and explanation graph for the fault scenario in Example 15.3.



(a) Object part of the base graph.

(b) Thread part of the base graph.

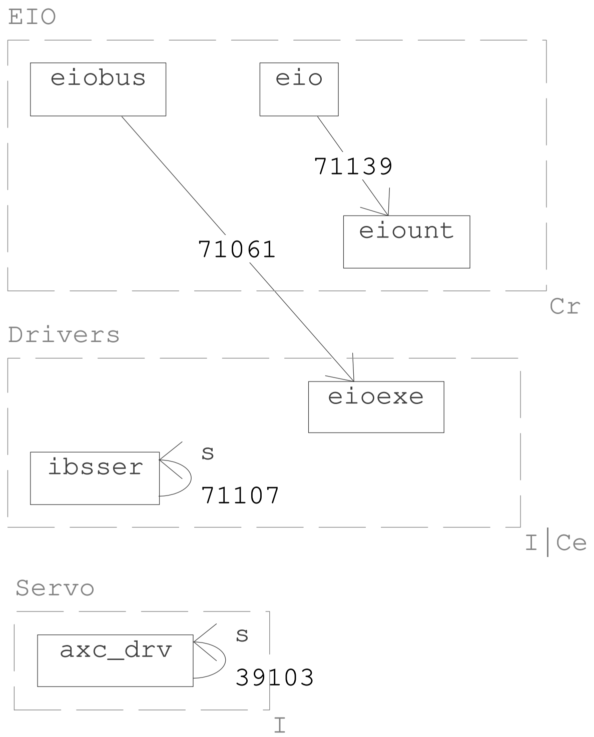
(c) The explanation graph.

Figure 15.6: The extended base graph and explanation graph for the fault scenario in Example 15.3.

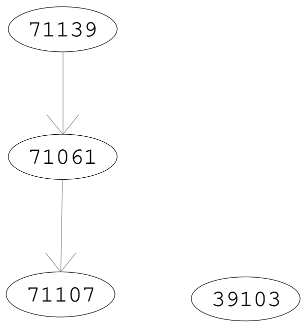
Example 15.4 Short-circuit
We present an example of a fault scenario where fault propagation has occured in the environment of the system, i.e., outside of the software.

During normal operation, a field-bus suffers a short-circuit, which gets reported both as a bus fault, but also as a fault in the power supply to the robot. The initial base graph and explanation graph are shown in Figure 15.7. As can be seen, the explanation graph gives two maximal error messages; one regarding the bus and one regarding the power supply. Since the same primary fault has manifested itself in two different parts of the system through physical dependencies not incorporated in the fault isolation scheme, this is the best result we can hope for.

The Algorithm 14.1 (ClassExtendGraph) connects the base graph better, similar to Example 15.1, but the two maximal elements of the explanation graph are the same. The other extension algorithms have no effect. □



(a) Base graph.



(b) Explanation graph.

Figure 15.7: The initial base graph and explanation graph for the fault scenario in Example 15.4.

Chapter 16

Conclusions, Part II

We have presented a general, scalable and easily maintainable fault isolation scheme aimed at the problem of fault propagation between software modules in a large-scale control system with object oriented architecture. The goal of the fault isolation scheme is in principle to pinpoint the most significant error message in a fault scenaro. The most significant error message is acquired as the maximal element of a cause-effect relation. A prototype implementation of the scheme, *DrRobot*, has been tested on several real fault scenarios from an industrial robot control system with object oriented architecture developed by ABB Robotics.

In the fault scenarios studied in this work, the fault isolation scheme, and the visualizations by *DrRobot*, provides insights into the scenarios and system that otherwise require a substantial experience. The correct maximal error message, or thread, is either pointed out in all cases, or the scheme is clearly inconclusive, i.e., the cause-effect relation is not connected or has several maximal elements. We hence believe that the proposed fault isolation scheme can be used to great advantage both to support end-users with fault isolation in production systems, and as a tool for system developers.

To further test the scheme in large scale, and make use of it in a production system, the signatures of all run-time error messages in the system must be determined and maintained. The system model must be accordingly extended, and maintained, to comprise the relevant parts of the system and clustering of the log must be addressed.

Bibliography

[1] S. B. Akers. Functional testing with binary decision diagrams. In *Eighth Annual Conference on Fault-Tolerant Computing*, pages 75–82, 1978.

[2] D.S. Arnon. A bibliography of quantifier elimination for real closed fields. *J. Symbolic Comput.*, 5(1-2):267–274, 1988.

[3] S. Bavishi and E. K. P. Chong. Automated fault diagnosis using a discrete event systems framework. In *Proceedings of the International Symposium on Intelligent Control*, pages 213–8. IEEE, 1994.

[4] M. Blanke. Consistent design of dependable control systems. *Control Engineering Practice*, 4(9):1305–12, 1996.

[5] M. Blanke, R. Izadi-Zamanabadi, S. A. Bøgh, and C. P. Lunau. Fault tolerant control – a holistic view. *Control Engineering Practice*, 5(5):693–70, 1997.

[6] G. Booch. *Object-Oriented Analysis and Design: With Applications*. Benjamin/Cummings, 2 edition, 1994.

[7] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th Design Automation Conference*, pages 40–45. ACM/IEEE, 1990.

[8] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[9] Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. Technical Report CMU-CS-92-160, School of Computer Science, Carnegie Mellon University, July 1992.

[10] Randal E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *International Conference on Computer-Aided Design (ICCAD)*, November 1995.

[11] Alan Burns and Andy Weddings. *Real-Time Systems and Programming Languages*. Addison-Wesley, second edition, 1996.

[12] J. P. Cassar, M. Staroswiecki, and P. DeClerck. Structural decomposition of large scale systems for the design of failure detection and identification procedures. *Systems Science*, 20(1):31–42, 1994.

[13] C.Bousson, P. Gaborit, and M. Ghallab. Situation recognition: Representation and algorithms. In *Proceedings of the 13th IJCAI*, 1993.

[14] Yi-Liang Cheng and G. Provan. Modeling and diagnosis of timed discrete event systems - a factory automation example. In *Proceedings of the American Control Conference ACC97*, pages 31–6, Albuquerque, New Mexico, June 1997.

[15] L. Console, L. Portinale, D. Dupre, and P. Torasso. Diagnostic reasoning across different time points. In *Proceedings of 10th European Conference on Artificial Intelligence ECAI 92*, pages 369–73, 1992.

[16] L. Console, L. Portinale, D. Dupre, and P. Torasso. Diagnosing time-varying misbehavior: an approach based on model decomposition. *Annals-of-Mathematics-and-Artificial-Intelligence*, 11(1-4):381–98, 1994.

[17] A. Darwiche. Model-based diagnosis using structured system descriptions. *Journal of Artificial Intelligence Research*, 8:165–222, 1998.

[18] S. R. Das and L. E. Holloway. Learning of time templates from system observation. In *Proceedings of the American Control Conference (ACC)*, volume 4, pages 2626–30, Seattle, Washington, June 1995. IFAC.

[19] J.H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra. Systems and Algorithms for Algebraic Computation.* Academic Press, 1988.

[20] J. de Kleer, A. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56, 1992.

[21] T. DeMarco. *Structured Analysis and System Specification.* Prentice-Hall, 1979.

[22] B. P. Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems.* Addison Wesley, 1998.

[23] M. S. Elliott. Computer-assisted fault tree construction using a knowledge-based approach. *IEEE Transactions on Reliability*, 43(1):112–20, March 1994.

[24] P. M. Frank. Advances in observer-based fault diagnosis. In *Survey Paper Tooldiagnosis '93 (International Conference on Fault Diagnosis)*, Toulouse,France, April 1993.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[26] Johann Gamper. *A Temporal Reasoning and Abstraction Framework for Model-Based Diagnosis Systems*. Phd thesis D82, RWTH, Aachen, Germany, July 1996.

[27] R. Germundsson. *Symbolic Systems - Theory, Computation and Applications*. Phd thesis 389, Department of Electrical Engineering, Linköping University, Linköping, Sweden, September 1995.

[28] Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics*. Addison Wesley, second edition, 1989.

[29] J. Gunnarsson. Algebraic methods for discrete event systems - a tutorial. In *Proceedings of the International Workshop on Discrete Event Systems 1996 (WODES96)*, pages 18–23, Edingburgh, Scotland, August 1996. IEE.

[30] J. Gunnarsson. *Symbolic Methods and Tools for Discrete Event Dynamic Systems*. Phd thesis 477, Department of Electrical Engineering, Linköping University, Linköping, Sweden, May 1997.

[31] J. Gunnarsson, J. Plantin, and R. Germundsson. Verification of a large discrete system using algebraic methods. In *Proceedings of the International Workshop on Discrete Event Systems 1996 (WODES96)*, pages 214–219, Edingburgh, Scotland, August 1996. IEE.

[32] Walter Hamscher, Luca Console, and Johan de Kleer. *Readings in Model-Based Diagnosis*. Morgan Kaufmann Publishers, 1992.

[33] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, pages 31–42, July 1997.

[34] D. M. Himmelblau. Fault detection and diagnosis in chemical and petrochemical processes. *Chemical Eng. Mon.*, 8, 1978.

[35] L. E. Holloway and S. Chand. Time templates for discrete event fault monitoring in manufacturing systems. In *Proceedings of the American Control Conference (ACC)*, volume 1, pages 701–706, Baltimore, Maryland, 1994. IFAC.

[36] Lawrence E. Holloway. On-line fault monitoring of a class of hybrid systems using templates with dynamic time scaling. In *Hybrid Systems III*, pages 258–69, New Brunswick, NJ, USA, October 1995. Springer-Verlag, Berlin, Germany.

[37] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[38] Watts S. Humphrey. *A Discipline for Software Engineering*. SEI Series in Software Engineering. Addison-Wesley, 1995.

[39] R. Germundsson J. Gunnarsson. Dynamic verification of a large discrete system. In *Proceedings of the 35th IEEE Conference on Decision and Control (CDC)*, New Orleans, USA, February 1996. IEEE.

[40] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[41] S. Lapp and G. Powers. Computer aided synthesis of fault trees. *IEEE Transactions on Reliability*, 26(1):2–13, April 1977.

[42] J. C. Laprie. Dependability: from concepts to limits. In *Proceedings of 12th International Conference on Computer Safety, Reliability and Security (SAFECOMP'93)*, pages 157–68, Berlin, Germany, 1993.

[43] J. C. Laprie, J. Arlat, C. Béounes, and K. Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, 23(7):35–51, 1990.

[44] M. Larsson. Diagnosis and analysis of diagnosis properties using discrete event dynamic systems. In *Proceedings of the 37th IEEE Conference on Decision and Control (CDC)*, Tampa, Florida, USA, December 1998. IEEE.

[45] M. Larsson, I. Klein, D. Lawesson, and U. Nilsson. Model based fault isolation for object-oriented control systems. Technical Report LiTH-ISY-R-2205, Dept. of Electrical Engineering, Linköping University, November 1999. Can be acquired at `http://control.isy.liu.se/publications/`.

[46] M. Larsson, I. Klein, D. Lawesson, and U. Nilsson. The need for fault isolation in object-oriented control systems. Technical Report LiTH-ISY-R-2098, Dept. of Electrical Engineering, Linköping University, February 1999. Can be acquired at `http://control.isy.liu.se/publications/`.

[47] Feng Lin. Diagnosability of discrete-event systems and its applications. *Discrete Event Dynaic Systems*, 4:197–212, 1994.

[48] L. Ljung. *System Identification: Theory for the User*. Prentice-Hall, 2nd edition, 1999.

[49] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[50] R. Miller and A. Tripathi. Issues with exception handling in object-oriented systems. In *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP97)*, Jyväskylä, Finland, June 1997.

[51] R. Milne, C. Nicol, M. Ghallab, L. Trave-Massuyes, C.Bousson, J.Quevedo C. Dousson, J. Aguilar, and A. Guasch. Tiger:real-time situation assessment of dynamic systems. *Intelligent Systems Engineering*, 1994.

[52] B. Mishra. *Algorithmic Algebra*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[53] NASA. The New Millenium Program. `http://nmp.jpl.nasa.gov/`.

[54] M. Nyberg. *Model Based Fault Diagnosis: Methods, Theory and Automotive Engine Applications.* Phd thesis 591, Department of Electrical Engineering, Linköping University, Linköping, Sweden, June 1999.

[55] Object Management Group. UML Notation Guide, version 1.1, 1997. Can be acquired at `http://www.rational.com/uml/`.

[56] Object Management Group. `http://www.omg.org`.

[57] D. N. Pandalai and L. E. Holloway. Condition templates: Improved distributed models for automated fault monitoring of manufacturing systems. In *Proceedings of the International Conference on Robotics and Automation*, pages 515–20, Minneapolis, Minnesota, April 1996. IEEE.

[58] R. J. Patton. Robust model-based fault diagnosis: The state of the art. In *Proceedings of the IFAC Symposium of Fault Detection, Supervision and Safety for Technical Processes SAFEPROCESS94*, IFAC Fault Detection, Supervision and Safety for Technical Processes, pages 1–24, Espoo, Finland, 1994.

[59] R. J. Patton. *Robust Model-Based Fault Diagnosis for Dynamic Systems.* Kluwer Academic Publishers, 1999.

[60] J. Plantin. *Algebraic Methods for Verification and Control of Discrete Event Dynamic Systems.* Licentiate thesis LIU-TEK-LIC-1995:33, Department of Electrical Engineering, Linköping University, Linköping, Sweden, June 1995.

[61] David Poole. Explanation and prediction: An architecture for default and abductive reasoning. *Computational Intelligence*, 5(2):97–110, 1989.

[62] Peter J.G. Ramadge. Observability of discrete event systems. In *Proceedings of the 25th IEEE Conference on Decision and Control (CDC)*. IEEE, December 1986.

[63] Peter J.G. Ramadge and W.Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[64] Rational Software Corporation. Rational Rose. `http://www.rational.com/rose/`.

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

[66] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[67] T. Ruokonen, editor. *IFAC Symposium of Fault Detection, Supervision and Safety for Technical Processes (SAFEPROCESS 94)*, Espoo, Helsinki, Finland, June 1994. IFAC.

[68] M. Sampath. *A Discrete Event Systems Approach to Failure Diagnosis*. PhD thesis, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, 1995.

[69] M. Sampath, S. Lafortune, and D. Tenektzis. Active diagnosis of discrete event systems. Technical Report CGR-97-06, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, USA, March 1994.

[70] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, September 1995.

[71] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Tenektzis. Failure diagnosis using discrete event models. Technical Report CGR-94-03, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, USA, May 1994.

[72] W. T. Scherer and C. C. White. A survey of expert systems for equipment maintenace and diagnostics. In S. G. Tzafestas, editor, *Knowledge-based system diagnosis, supervision and control*, pages 285–300. Plenum, New York, 1989.

[73] Robert Sedgewick. *Algorithms*. Addison Wesley, second edition, 1988.

[74] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley, 1994.

[75] Arvind Srinivasan, Timothy Kam, Shrad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *1990 IEEE International Conference on Computer-Aided Design*, November 1990.

[76] S. Tzafestas and K. Watanabe. Modern approaches to system/sensor fault detection and diagnosis. *Journal A*, 31(4):42–57, December 1990.

[77] N. Viswanadham and T. L. Johnson. Fault detection and diagnosis of automated manufacturing systems. In *Proceedings of the 27th IEEE Conference on Decision and Control (CDC)*, volume 3, pages 2301–6, Austin, Texas, 1988. IEEE.

[78] R. D. Vries. An automated methodology for generating a fault tree. *IEEE Transactions on Reliability*, 39(1):76–86, April 1990.

[79] B. Williams and P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96) Building Construction*, pages 971–978, 1996.

[80] Alan S. Willsky. A survey of design methods for failure detection in dynamic systems. *Automatica*, 12:601–11, 1976.

[81] Wolfram Reasearch. Makers of *Mathematica*. `http://www.wolfram.com/`.

[82] Cüneyt M. Özveren and Alan S. Willsky. Observability of discrete event dynamic systems. *IEEE Transactions on Automatic Control*, 35(7):797–806, July 1990.

Index