Model Based Fault Isolation for Object-Oriented Control Systems

M. Larsson, I. Klein  D. Lawesson, U. Nilsson
Dept. of Electrical Engineering  Dept. of Computer and Info. Science
Linköping University  Linköping University
S-581 83 Linköping, Sweden  S-581 83 Linköping, Sweden
magnusl, inger@isy.liu.se  danla, ulfni@ida.liu.se

10 November 1999

Report no.: LiTH-ISY-R-2205

Technical reports from the Automatic Control group in Linköping are available for download at http://control.isy.liu.se/publications/.
Model Based Fault Isolation for Object-Oriented Control Systems

M. Larsson, I. Klein  
Dept. of Electrical Engineering  
Linköping University  
S-581 83 Linköping, Sweden  
magnusl, inger@isy.liu.se

D. Lawesson, U. Nilsson  
Dept. of Computer and Info. Science  
Linköping University  
S-581 83 Linköping, Sweden  
danla, ulfni@ida.liu.se

ABSTRACT

This report addresses the problem of fault propagation between software modules in a large industrial control system with an 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 thus causing problems for system operators and service personnel trying to quickly isolate the real fault. A system developer with insight in the system design, can, of course, often easily interpret the multitude of error messages from a fault scenario and isolate the primary cause. The key observation is that this can often be done using high-level models of the system and the fault propagation. We have made an effort to automate this procedure, and we 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 outline 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. The approach is illustrated by means of several examples. The approach has been implemented and tested on a commercial control system for industrial robots developed by ABB Robotics. A patent claim has also been filed with the Swedish Patent Office (PRV).

Keywords: fault isolation, control system design, UML.
Chapter 1
Introduction

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 [7]). Today, many large systems are designed using an object oriented approach, see e.g., [8, 22, 25]. This has several advantages over traditional approaches, including better possibility to cope with complexity and to facilitate maintenance and reuse (see e.g. [3]). However, this leads to new kinds of problems; in this report 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 object oriented goal to encapsulate information implies that individual objects or groups 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 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 fault to an operator, so that the fault can be quickly repaired and normal operation restored. To present the operator with a multitude of error messages from different parts of the system is not a very desirable behavior. The problem from the operators point of view is schematically illustrated in Figure 1.1.

Using traditional software development methods, it is formally possible to have the state of the whole control system known centrally. It is possible, at least in principle, 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 1.1 how these object-oriented design goals often stand in direct conflict with the need to generate concise information about a fault scenario.
As basic inspiration and case study, 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 2.3.

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 basically consists of deriving 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 1.2.

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
Figure 1.2: Main flow diagram of the fault isolation procedure
An example from the ABB Robotics industrial robot control system is demonstrated in Section 1.3 as a preview of the fault isolation scheme, and the implementation and practical results are more thoroughly presented in Chapter 6. A patent claim has been filed with the Swedish Patent Office (PRV) [17].

1.1 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. A preliminary discussion on these problems can be found also in [16].

In our case there are two main types of run-time faults that occur: hardware faults and real-time faults. The real-time faults are due to several reasons to be discussed in Chapter 4, 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 what follows.

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 an 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 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 what follows.

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 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.

\(^1\)With system we will mean the control system, if not otherwise clear from context.
**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 available 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 here takes a more abstract view, and addresses the problem, mainly fault propagation, from above. The methods described here 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 [18], 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.

### 1.2 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 quickly determine the root cause of a fault by studying the logged error messages from a fault scenario. This is of course not very surprising, but it is valid also for fault scenarios that causes fault isolation problems for end users. This report describes 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 1.1, then one fault isolation method of course is to compile a database, or to use an expert system [23, 26], 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 development 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 not consider these associative model structures further, but focus on the use of so called explanatory, or deep models.

The fault propagation in a software system as described in Section 1.1 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., [1, 2].

The behavior of software components does not follow 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 [10] (see also Section 2.2.4), 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 of
[15] for fault isolation. But such methods 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 compo-
nents 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 development
process, at least not in today’s practice. The objections against such 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 here is to develop a model based fault isolation method that
is a natural part of the software development 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 them and use
them for other purposes as well. We propose a structural scheme for fault isolation,
where error messages are explained locally, using the information available to 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 development 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 mes-
  sage 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 to the
operator. When 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”.
The main parts of the proposed fault isolation scheme are visualized in Figure 1.3.
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 1.3. To
the authors’ 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 unexpe-
rienced 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.

In order for our system model to be supported by the software development
process, we have chosen to use the Unified Modeling Language (UML); it has
become a de facto standard modeling and development tool for object oriented systems. The system model used for fault isolation is then an integral part of the system documentation (see Section 2.2 and, e.g., [19, 8]). 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 development 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 2.2. 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 3).

**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 or 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 object oriented system is regarded as a collection of collaborating, fairly intelligent, but narrow-minded individuals, these three types can be characterized by the statements “I did it”, “he did it” respectively “I didn’t do it”.

Even though it might seem tempting just to pick 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 may 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 4, 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 available. 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 are still required to make an effort to inhibit the system from sending unnecessary error messages.

### 1.3 Example

We illustrate our approach by going 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 follow the scheme outlined in Figure 1.3.

\(^2\)In UML-terms this will be a *package*. 
Figure 1.4: The relevant part (cluster) of the message log for the fault scenario in Section 1.3.

In this 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, which 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 1.4. In general the message log contains 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 1.3.

The set of error messages can, even after clustering, be hard to interpret without insight in 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 this is not necessarily true in general (see Chapter 4). It may happen that there are other messages that are not the result of fault propagation, but correspond to real faults.

As explained in Section 1.2, an error message is sent by an object and can be interpreted either as a complaint on another object (in which case the object is a complainer and the misbehaving object is called the complainee), or a statement that something is wrong with the object itself. We call this information the error message signature (cf. Figure 1.3), 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 1.4 is shown in Figure 1.5(a). Edges between pairs of nodes denote relational error messages and should be read “complains on”. The self-loop adorned with \textit{int} in the figure is used to denote an internal error message (Section 1.2).

The arrow with the triangular head between two of the nodes in Figure 1.5(a) is the UML notation for generalization (see Chapter 2), and basically means that the two nodes in the base graph should be interpreted as the same instantiated object.

Note that the two last of the original six messages in the cluster in Figure 1.4 are not present in the base graph in Figure 1.5(a). These messages are so-called \textit{operational} messages; that inform the user of major state changes in the system. In this case the operational messages state that the program execution has reached a failure state and is aborted. Such 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 1.2. 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.

When looking at the base graph in Figure 1.5(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 1.3). According to this assumption, we can construct the base graph directly from a cause-effect relation (also called explanation relation) on the error messages, where an error message is related to the error messages that 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 1.5(a) is shown in Figure 1.5(b).

As can be seen in Figure 1.5(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 the fact that the initial base graph in Figure 1.5(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 1.6(a) (cf. Figure 1.3). The added (derived) “complains on” edge in the base graph is dotted, 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 1.6(b) has a unique maximal element; the one stating that the CAN-bus has failed.

1.4 Model classification

In Section 1.2, a fault isolation scheme aimed at the problem described in Section 1.1 was outlined. In this section, we try to put the outlined scheme in perspective, by discussing the information sources available for a model-based automated fault isolation scheme aimed at the problem described in Section 1.1. We also propose a simple classification of the available information sources. Our focus is on deep models and model-based approaches, for reasons discussed in Section 1.2.

The fault isolation scheme presented here 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 1.7. A static structure model contains
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 2.2.3) 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 here, but digraphs have been used for this purpose in the context of analytical redundancy methods, see, e.g., [5]. For the use of structure models in AI model-based diagnosis, see, e.g., [6].

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 2.2.3. 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 4.

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 2.2.

For a specific fault scenario (see Chapter 4) 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.

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 1.8. 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

---

\(^3\)Strictly speaking the software does not dump core but rather suspends. However, the suspended system can be conceptualized as a core dump.
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 be-
tween 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 soft-
ware. Apart from error messages, the log also contains limited information about
state transitions of major software entities. The log is discussed in detail in Sec-
tion 4.2.1.

1.5 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 implies less dependence on the system model, but
it also implies more work 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 4 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 the requirement 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 instan-
tiations in the run-time system.

- Too general super-classes in the system model are avoided.

Even if the complete run-time object structure of the system often is very dynamic
and constantly changing, there are usually 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.
Some design patterns, such as the recursive Composite pattern (see [9]) have a run-time object structure which is dramatically different from the static class structure. Such patterns are rare in today’s practice, but as the use of patterns gets more common, the issue needs to be addressed in the future.

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 inspired this work in the first place. The required properties of the system and system model are further discussed in Chapter 4 and subsequent chapters.

There is a wide class of faults that are not dealt with in the suggested approach, and have 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 common. However, during this phase, experts on the system are present, and these faults are therefore not considered further here.

Faults in the design and software bugs are also neglected 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. [13, 14].

1.6 Outline of the report

In Chapter 2, 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. Chapter 3 formalizes the system model and describes the part of UML that is used in the fault isolation scheme. Chapter 4 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 5. The implementation and several examples from the ABB Robotics control system are demonstrated in Chapter 6.
Chapter 2

Background

2.1 Object orientation: a brief motivation

This section gives a brief motivation for and introduction to object oriented methods in software development. For a thorough treatment, see e.g., [22].

The traditional structured software development methods (see, e.g., [7]) 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 artifacts, 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 improved 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. In the sequel we will 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 object-oriented method scale better to large 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 is often 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 object-oriented development is constructed around the main artifacts 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., [4] 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 themselves 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 2.2.3. 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 a 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 2.2.1.

Inheritance and associations will be further discussed in the framework of the Unified Modeling Language in Section 2.2.

2.2 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) [20]. The OMG
• Use case diagram
• Class diagram
• Behavior diagrams
  – Statechart diagram
  – Activity diagram
  – Interaction diagrams
• Implementation diagrams
  – Component diagram
  – Deployment diagram

Table 2.1: 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 [19], 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.

Lately the UML has 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 are also emerging; presently the most well known is Rational Rose™ [21]. Rational Rose™ is a tool for modeling, design and documentation of object oriented systems, and can be a well-integrated part of the software development 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 6.1.

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 [19], or a more accessible book on the subject, e.g., [8]. The subset of the UML used in the fault isolation approach is formally specified in Chapter 3.

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 available to the user. The available 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 2.1.

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 pre
defined icons. We will use stereotypes in this work to model certain asynchronous
communication schemes that are 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 deploy-
ment 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 2.2.1 we discuss some general issues with concurrently executing
threads. The different UML diagrams in Table 2.1 are discussed in Section 2.2.2
to Section 2.2.5.

2.2.1 Concurrency

A software system often has several concurrently executing threads that communi-
cate 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 Sec-
tion 2.2.3 about class diagrams. We also need so called task diagrams, that ex-
plicitly describe the run-time collaboration of threads (see, e.g., [8]). This will be
discussed and demonstrated in Section 2.2.5.

Communication between threads can take on many forms, both regarding the
scheme on a conceptual, logical level and regarding implementation. The possi-
ble implementations include asynchronous mail via the operating system, a syn-
chronization of several threads (a rendezvous), shared memory areas and Remote
Procedure Calls (RPC) in a distributed system, see e.g., [4].

The implementation issues are not of particular interest to us, and we will as-
sume 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 in-
terest 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.

- 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 2.1 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 4. See further Section 2.2.3 and Section 2.2.5 for modeling of thread collaborations.

2.2.2 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., [12]. 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.

2.2.3 Class diagram

Class diagrams in the UML are a melding of the class diagrams of OMT, Booch, and most other object oriented methods (see [19] for a more extensive story). In UML class diagrams, classes (Section 2.1) 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 2.2.
Figure 2.2: The UML notation for a class called Sensor, with attributes and methods.

Figure 2.3: An example of the inheritance notation in the UML.

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 triangular head, see Figure 2.3. 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 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 2.3.

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 2.4. 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 2.4, 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.

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 2.5. The difference from Figure 2.4 is that the sensors are now
considered to be a part of the controller instead of an 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 want 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 2.5.

The Enqueue- and Subscribe-pattern patterns for asynchronous thread collaboration as described in Section 2.2.1 are modeled with stereotypes in UML class diagrams as is shown in Figure 2.6. 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 2.7). Packages themselves can have dependencies between them. The notation for a package is a folder shaped icon as in Figure 2.7, where also the notation for package dependency is shown. This type of dependency means 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.

### 2.2.4 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.

The *statechart diagram* is based on the statecharts formalism of David Harel [10] 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.
Figure 2.7: Example of packages and package dependency.

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 [7].

Interaction diagrams are typically 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.

2.2.5 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 [19], this information should be depicted in the deployment diagram. Unfortunately, the UML tool we use, Rational Rose™ [21], does not 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 2.6, is shown in Figure 2.8. 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 2.8 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.

2.3 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, gluing, 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 2.9.

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 ex-
ternal equipment can be controlled by RAPID programs from the control system,
that in turn can communicate with, e.g., PLCs and be connected to local networks.
The control system is multi-threaded, i.e., there are several concurrently execut-
ing tasks, on several processors. The threads communicate both asynchronously
and synchronously via an IPC messaging facility. The communication patterns
subscription and enqueuing described in Section 2.2.1 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 3

System model

The model structure used in the fault isolation scheme is a subset of the UML, and in this chapter we attempt to formalize exactly what parts of the UML model, as presented in Section 2.2, that are used. Our system model consists of two separate parts, a class model and a thread model.

3.1 Class model

The class model used, that actually also contains package information, can be formalized as the five-tuple

\[ < \text{Classes}, \rightarrow, \text{A}, \text{PC}, \text{PD} > . \]

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 runtime system contains many instantiated objects from Classes.

The inheritance, or generalization, relation between classes in the UML is present in our system model as the transitive and anti-symmetric relation

\[ c \rightarrow c' \in (\text{Classes} \times \text{Classes}). \]

C.f. the UML notation in Figure 3.1. 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 Classes → 2^Classes, where 2^Classes denotes the set of all subsets of Classes.

\[
\begin{align*}
\text{desc}(c) &= \{ c' \in \text{Classes} \mid c' \rightarrow c \} \cup \{ c \} \\
\text{asc}(c) &= \{ c' \in \text{Classes} \mid c \rightarrow c' \} \cup \{ c \} \\
\text{rel}(c) &= \text{desc}(c) \cup \text{asc}(c)
\end{align*}
\]

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 \( \text{rel}(c) \) ("relatives" of \( c \)), are illustrated in Figure 3.2.

The reflexive and symmetric closure of the relation \( \rightarrow \) is an equivalence relation, and by \( [c] \) we denote the equivalence class\(^1\) that contains \( c \). Note that \( \text{rel}(c) \) and \( [c] \) in general are different, e.g., in Figure 3.2, \( [c] \) consists of all five classes, including the sibling of \( c \).

The association and aggregation relationship in the UML is represented with the association relation

\[
A \colon \mathcal{C} \to A \subseteq (\mathcal{C} \times S_A \times \mathcal{C}).
\]

\( S_A \) 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 \( <<\text{Enqueue}}>> \) and \( <<\text{Subscribe}}>> \) (see Section 2.2.3). 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 5, we show how this is used in the fault isolation.

For later use, we introduce selector functions for the association relation. Let \( a = c \mapsto s \in A \), then

\[
\text{client}(a) = c \\
\text{server}(a) = s \\
\text{stereotype}(a) = t.
\]

The logical partitioning of classes into Packages in a UML model is represented with the package partition \( \mathcal{P}_C \subseteq 2^{\mathcal{C}} \), where all elements in \( \mathcal{P}_C \) are disjoint.

The inheritance relation \( \rightarrow \) and the package partition \( \mathcal{P}_C \) are restricted so that each equivalence class \( [c] \subseteq \mathcal{C} \) is part of exactly one package, i.e., \( [c] \subseteq p \in \mathcal{P}_C \).

\(^1\) An unfortunate terminology conflict between set theory and object orientation!
The package dependencies in the UML model will also be used for the fault isolation, and we hence define the \textit{package dependency relation} PD 

\[ c \rightarrow s \in \text{PD} \subseteq (P_C \times P_C). \]

For later use, we introduce the corresponding selector functions. Let \( d = c \rightarrow s \in \text{PD}, \) then

\[ \text{client}(d) = c, \quad \text{server}(d) = s. \]

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 \textit{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.
3.2 Thread model

The thread part of the system model is simpler than the class model, and simply is a pair with thread components and stereotyped thread dependencies

\[
< T_C, TD >.
\]

\( T_C \) is the set of all threads modeled as components in the UML model as described in Section 2.2.5, and \( TD \) is the set of thread dependencies

\[
\text{ct} \overset{t}{\rightarrow} \text{st} \in TD \subset (T_C \times S_{TD} \times T_C).
\]

\( S_{TD} \) 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 \(<\text{Enqueue}>\) and \(<\text{Subscribe}>\) (see Chapter 2). In Chapter 5, we show how this is used in the fault isolation.

The selector functions for a thread dependency relation \( d_t = \text{ct} \overset{t}{\rightarrow} \text{st} \) are defined as

\[
\begin{align*}
\text{client}(d_t) &= \text{ct} \\
\text{server}(d_t) &= \text{st} \\
\text{stereotype}(d_t) &= t.
\end{align*}
\]
Chapter 4

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.

We first specify what we mean with a fault in the context of control systems.

**Definition 4.1 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 1, 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 4.2 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 what follows, 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 occurring 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) typically just a couple of seconds and very rarely exceeding 10 minutes (see further Section 4.3). However, most ideas presented here are applicable also in a multiple fault situation, as is briefly discussed in Section 5.3. The single fault that has occurred in a fault scenario will also be called the primary fault.
In Chapter 1, 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.

Our approach to fault isolation is based on some basic assumptions and facts about 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 4.1. The formal classification of error messages and the definition of error message signatures according to the explanation models are presented in Section 4.2.1. The clustering of the system log into fault scenarios has not yet been implemented, but the issue is discussed in Section 4.3. The formal representation of a fault scenario, the base graph, used in the fault isolation scheme is introduced and formally defined in Section 4.4.

4.1 Explanation models

In this section we 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 5.3, 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.

4.1.1 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 2.2.3. Faults (Definition 4.1) 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 4.2.1.
For closely collaborating objects it is often possible to suppress error messages by negotiations between the objects, 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”, there are typically 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”, it 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 4.2.2.

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 4.1.2. The explanation model needed to include thread and concurrency concepts in the fault isolation is discussed in Section 4.1.3.
4.1.2 Object explanation model

Many instantiated objects collaborate and depend on each other as described in Section 2.1 and Section 2.2.3. In a well designed system, the objects (classes) and modules, logically partitions the system into different areas of responsibility. The basic object explanation model simply is 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 1.3, if the problems one object experiences can be related to problems in another object, the first is assumed to be 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 1.2).

**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 2.3. 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 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 the 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 4.2.1.

A natural approach in the case of ordinary method calls, is to put information about the error condition in the complainee in the returned error code. There are several drawbacks associated with this. The interface between objects gets quite complicated and unwieldy and hence hard to manage. More importantly perhaps, 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 about what other system entity failed to perform as expected. This can be due to encapsulation and modularity reasons.

**Package explanation model**  In the object explanation model, we also include the assumption 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 5.3.

4.1.3 Thread explanation model

One dimension of the fault isolation task that cannot be overlooked is 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 is twofold:

- As discussed in Section 2.2.1, 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 4.1.2.

- The problem a thread experiences can cause error conditions in objects executing on the thread.

Examples will be given in Section 4.4 and Chapter 6.

A thread of control may slow down, get stuck or exit abnormally for a number of reasons, all of which can initiate error conditions for other threads and 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 4.2.1.

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.

\(^1\)In a well designed system, that is.
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 be deadlock situations or not properly tuned timeout lengths and task priorities. A thread may exit abnormally due to operating system violations, 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 low 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 2.2.1.

If two threads communicate synchronously, error conditions in the form of time-outs 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 2.2.1), 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 to make a distinction related to how the thread collaboration is modeled in the task diagram of the system model (see Section 2.2.1 and Section 2.2.5). In the task diagram (see Figure 2.8), 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., [8]) 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 4.2.3). However, in the current implementation, all IPC-error messages are assumed to be relational.

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 5.
4.2 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 available also after a shutdown of the system.

Definition 4.3 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 2 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 4.3. 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 chronological 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 to be useful, is that faults should be reported as close to the source as possible, where also the best information on the occurred 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 4.1.1 and the information they need to contain, are treated in detail in Section 4.2.1. Operational and supervisory messages are often sent to the log during a fault scenario and are discussed in Section 4.2.2 and Section 4.2.3 respectively. Section 4.2.4 contains a brief discussion of unobservable faults in our context.

4.2.1 Run-time error messages

The messages sent to the log during the “upwards” fault propagation discussed in Section 4.1.1 will be called run-time error messages. The error messages constitute
- Object error messages
  - Internal
  - Relational
    - Known complainee
    - Unknown complainee.
- Thread (IPC) error messages
  - Relational
    - Known complainee
    - Unknown complainee
    - Direction server
    - Direction client

Table 4.1: The six error message classifications according to Section 4.1.

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 1.4).

**Definition 4.4 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 4.1.1).

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 4.1. We also specify the corresponding information an error message must contain for fault isolation purposes.

According to the discussion in Section 4.1, the error messages are classified into six types, shown in Table 4.1. In Chapter 1 we only mentioned three groups, but now we also take the IPC-error messages into account. The object error messages in Table 4.1 will also be called **ordinary error messages**.

To construct 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 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 1.2.
- 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 4.4.

In Figure 4.1 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 2.2.3). Some of the messages sent to the system log that are of interest to partition the log into clusters corresponding to fault scenarios (Section 4.3) are not represented in Figure 4.1. These messages are discussed in Section 4.2.2. Also recall the example in Section 1.3.

**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 4.4 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 the corresponding entities in the system model in Chapter 3. 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 4.1. For a relational error message (class Relational in Figure 4.1), the complainee object is identified by 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 4.4, we will consider the sending thread to correspond to the message and not the complainer object. If an error condition arises when objects on different threads communicate, that error condition should be classified as an IPC-error condition or a relational unknown.
Figure 4.1: A possible error message signature design.

**Class IPC**  The class IPC in Figure 4.1 corresponds to the IPC-error conditions discussed in Section 4.1.3. 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 4.1.3. 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 are used, as discussed in Section 4.1.3.

**Instance versus class information**  The information in the error message signature can be divided in two: 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 3.1. 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 $T_C$ in the system model in Section 3.2. In the current implementation only class-level and thread type-level information is used in the error message signatures.

As mentioned in Section 1.5, 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 resolving 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 4.4, we will introduce the formal representation of a fault scenario in the form of a so-called base graph. The nodes of the base graph correspond to the objects and threads that have manifested themselves in the fault scenario, and the edges are the corresponding error messages. 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 4.4.

An instantiated object is described in the the system model (Chapter 3) 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 5, 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 4.4 for an object/class identified by an error message signature (attributes Complainer and Complainee in Figure 4.1).

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 2.2.5) 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 SendingThread) a-posteriori. The thread view of the fault scenario is further discussed in Section 4.4.

4.2.2 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 are 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 1.3.

However, most of these operational and status messages are irrelevant for the fault isolation. As described in Section 4.1.1; 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 focuses on the “real” error messages, discussed in Section 4.2.1, 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 4.3. However, if the fault isolation approach is extended to take more dynamic behavior into account, the operational and status messages can probably be utilized. This is a matter of ongoing and future research.

4.2.3 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 4.3. The potential usefulness of one of these messages in the analysis of a fault scenario, as part of the base graph in Section 4.4, 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 5.

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 4.1.3. These messages can be designed and classified as internal and contain information very useful to the user.

4.2.4 Unobservable faults

With an unobservable fault, we mean a fault which eventually causes the system to halt, where there is no directly corresponding error message; i.e., no error message that is most relevant in a natural way.

Faults may 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 dispatched, 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 1.4), and could in principle be included in the fault isolation procedure. The current implementation does not take the configuration into account.

Second, there may simply not be an error message directly corresponding to the occurred 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.

4.3 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 occurred 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 addressed in-depth 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 4.2.1). 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 what follows we will assume that the message log can be clustered properly into fault scenarios.

4.4 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 occurred error condition propagation in the fault scenario. In Chapter 5 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 4.4.3 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 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 4.2.1, 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 \(\text{asc}(c)\) (notation from Chapter 3).

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 \(\rightarrow\) from Chapter 3 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
Assumption 4.1 For each equivalence class $[c] \subseteq \text{Classes}$, there is at most one instantiated object in the run-time system that manifests itself in a specific fault scenario.

Assumption 4.1 gives us a design rule for the system model to be useful for fault isolation, that was mentioned also in Section 1.5: 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 4.4.1
- A data structure suitable for implementation, Section 4.4.2
- A formalism used to state algorithms and prove properties, Section 4.4.3

To analyze the base graph, and to decide if and how the base graph should be extended in Chapter 5, 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 4.4.4.

### 4.4.1 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 4.4.3 below), only differently visualized.

In Figure 4.2 we show the object view graphical notation, i.e., the Relational and Internal error messages (Table 4.1, Section 4.2.1) 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 \(rel_a\). 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 2.2, which have the semantic “uses”.

The objects are partitioned into packages in the UML system model, as described in Section 2.2.3, 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 4.1 Field bus fault scenario
The fault scenario is similar to the fault scenario in Section 1.3. 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 4.3. 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 4.4.4.

Note the first and last message in the log cluster. They are operational messages (Section 4.2.2), and can serve as divider events as described in Section 4.3. The “real” run-time error messages as described in Section 4.2.1 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, (\texttt{RealInstruction} and \texttt{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 4.1.

Assumption 4.2 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 5. As Assumption 4.1, 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 4.1 and Figure 4.1) is, as all error messages, sent by an object. In Section 4.2.1, Table 4.1, 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 \(\text{relT}_{\text{s}}\), \(\text{relT}_{\text{u}}\) and \(\text{relT}_{\text{C}}\) respectively. See Figure 4.4(a) and 4.5(a).
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 ID 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 4.3: The base graph and system log from the real fault scenario in Example 4.1.
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 4.4(a). For the relational IPC-error messages, both the complainer and complainee thread is shown. As discussed in Section 4.2.1, 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 4.4(b) and 4.5(b).

**Example 4.2 IPC fault scenario**
The fault scenario in this example is from a real situation, where the control system 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 4.4. Since the cluster only consists of three messages, the base graph is quite small. Using the assumption in Section 4.1.2, that error conditions can propagate within packages, the natural conclusion from the base graph is that the problem lies with Task 3. In Section 5.3, we will show how this conclusion is formally reached, and the example is continued in Chapter 6, Example 6.2.

This conclusion is as far as DrRobot can go, but since the original error messages as shown in Figure 4.4(c) were quite cryptic, this information strongly focuses the further fault isolation.

**Example 4.3 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 2.2.1).

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 4.5. In Example 6.3, Chapter 6, we will use the methods introduced in Chapter 5, and show how both the object part and thread part of this base graph
A

B

C

Package 1

Package 2

Task 1

Task 2

Task 3

(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 4.4: The object and thread parts of the base graph for the fault scenario in Example 4.2.
Figure 4.5: The object and thread parts of the base graph for the fault scenario in Example 4.3 can be extended using the system model, and how this extension can be used to get a clearer interpretation of the fault scenario.

4.4.2 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 4.6. 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 4.4.1, the semantic for an edge in a base graph is different than for a link in an object diagram, though. The normal 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 4.1, Section 4.2.1 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 2.2.3. The package-links, represented in Figure 4.6 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 4.1. The method NodeCl(), specified for the base graph nodes in Figure 4.6(b), will compute the classification of the node, to be defined in Section 4.4.4.
(a) The base graph elements.

(b) Detailed relationships between the base graph elements.

Figure 4.6: The Base Graph data structure.
4.4.3 Formal representation

For the purpose of stating algorithms and proving properties in Chapter 5, 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 4.6(b) could be useful to have in mind.

By comparing with Figure 4.6(b), it should not come as a surprise that a base graph formally consists of a six-tuple

\[ < O, P_O, T_O, O_L, P_L, T_L >. \]  \hspace{1cm} (4.1)

All six elements will be defined and explained below. The first three elements are the sets

\[ O \quad \text{— generalized objects} \]
\[ P_O \quad \text{— packages (a partition of } O) \]
\[ T_O \quad \text{— threads} \]

Each generalized object \( o \in O \) belongs to exactly one class \( c \in \text{Classes} \) in the system model defined in Chapter 3. We define the mapping class \( : O \rightarrow \text{Classes} \) as

\[ \text{class}(o) = c \in \text{Classes}. \]

Under Assumption 4.1, the mapping is injective.

The base graph package partition \( P_O \) is defined by the system model package partition \( P_C \). For \( o_1, o_2 \in O \), we have

\[ o_1, o_2 \in P_O \text{ iff class}(o_1), \text{class}(o_2) \in p_c \in P_C. \]

It follows immediately that each partition in \( P_O \) corresponds to a unique package in \( P_C \). With \( p_c \) and \( p_o \) as above, we define the mapping \( pmap : P_O \rightarrow P_C \) as

\[ \text{pmap}(p_o) = p_c. \]

This mapping is of course always injective, regardless of Assumption 4.1.

Each thread \( t_o \) that has manifested itself in the fault scenario, i.e., \( t_o \in T_O \), corresponds to exactly one thread \( t_c \in T_C \) in the system model. We define the mapping \( tmap : T_O \rightarrow T_C \) as

\[ \text{tmap}(p_o) = p_c \in T_C. \]

Under Assumption 4.1, the mapping is injective.

The three link sets, or edge sets in the base graph six-tuple (4.1) are

\[ o_1 \leftrightarrow o_2 \in O_L \subset (O \times C_{OL} \times T_O \times O) \quad \text{— Object links} \]
\[ p_1 \leftrightarrow p_2 \in P_L \subset (P_O \times C_{PL} \times P_O) \quad \text{— Package links} \]
\[ t_1 \leftrightarrow t_2 \in T_L \subset (T_O \times C_{TL} \times O \times T_O) \quad \text{— Thread links}. \]

The complete collection of link classifications are

\[ C_{OL} = \{ \text{int, rel, rel}_u, \text{rel}_d \} \]
\[ C_{PL} = \{ \text{obj, pack} \} \]
\[ C_{TL} = \{ \text{rel}_T, \text{rel}_{T^C}, \text{rel}_{T^S}, \text{rel}_{T_d}, \text{int}_{T_d} \}. \]  \hspace{1cm} (4.2)

55
All classifications except rel\(_{\text{der}}\), pack, relT\(_{\text{der}}\) and intT\(_{\text{der}}\) have been discussed in Section 4.2.1. 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, \(o_1 \xrightarrow{\text{rel}_t} o_1\) and \(o_1 \xrightarrow{\text{int}_t} o_1\).

The relational derived classifications, rel\(_{\text{der}}\) for object links, relT\(_{\text{der}}\) 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 5.

The internal derived classification for thread links, intT\(_{\text{der}}\), is used only when constructing the cause-effect relation in Section 5.3. As discussed in Section 4.1.3, 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 5.3 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 4.2 and 6.2).

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\(_{\text{der}}\), between objects in different packages. A package link \(p_1 \xrightarrow{\text{obj}} p_2\) is hence a member of PL if and only if there exists an object link \(o_1 \xrightarrow{\text{c};t} o_2 \in \text{OL}\) such that

\[
\text{c} \in \{\text{rel}, \text{rel}_\text{der}\}, \; o_1 \in p_1 \text{ and } o_2 \in p_2.
\]

The obj package links are hence redundant information, but are present as syntactic sugar to make the formulation of the extension algorithms in Chapter 5 easier and to stress the relationship between \(\langle [O], \text{OL} \rangle\) and \(\langle P_O, \text{PL} \rangle\). Package links with classification pack are added to PL using the system model, as further explained in Chapter 5.

Given the object link \(ol = o_1 \xrightarrow{\text{c};t} o_2 \in \text{OL}\), we define the selector functions:

\[
\begin{align*}
\text{cr}(ol) & = o_1 \quad \text{— complainer} \\
\text{ce}(ol) & = o_2 \quad \text{— complainee} \\
\text{cl}(ol) & = c \quad \text{— object link classification} \\
\text{th}(ol) & = t \quad \text{— sending thread}.
\end{align*}
\]

Corresponding selector functions for the thread link \(tl = t_1 \xrightarrow{\text{c};o} t_2\) are

\[
\begin{align*}
\text{cr}(tl) & = t_1 \quad \text{— complainer (i.e., sending) thread} \\
\text{ce}(tl) & = t_2 \quad \text{— complainee thread} \\
\text{cl}(tl) & = c \quad \text{— thread link classification} \\
\text{obj}(tl) & = o \quad \text{— sending object}.
\end{align*}
\]

The selector functions for a package link \(pl = p_1 \xrightarrow{\text{c}} p_2\) are then of course

\[
\begin{align*}
\text{cr}(pl) & = p_1 \quad \text{— complainer package} \\
\text{ce}(pl) & = p_2 \quad \text{— complainee package} \\
\text{cl}(pl) & = c \quad \text{— package link classification}.
\end{align*}
\]
The base graph is initially built up using error messages from a cluster of the system log. Each error message corresponds, in the obvious way, to an object link $o_1 \rightarrow o \in \text{OL}$ where $c \in \{\text{int, rel, rel}_r\} \subset \text{C OL}$, or a thread link $t_1 \rightarrow t_2 \in \text{TL}$ where $c \in \{\text{relT}, \text{relT}_u, \text{relT}_u\} \subset \text{C TL}$. We let $M$ be the set of all links corresponding to error messages involved in a base graph, that is, $M \subseteq \text{OL} \cup \text{TL}$. Initially we have $M = \text{OL} \cup \text{TL}$.

The inheritance relation for classes extends to generalized objects in the natural manner:

$$o \rightarrow o' \text{ iff class}(o) \rightarrow \text{class}(o')$$

where $o, o' \in \text{O}$. 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 3 also extends to generalized objects:

$$\text{rel}(o) = \{o' \in \text{O} | o \rightarrow o' \text{ or } o' \rightarrow o\}$$

and given the reflexive and symmetric closure of the inheritance relation, $[o]$ denotes the equivalence class that contains $o$. We also denote the set of equivalence classes in $\text{O}$ as $[\text{O}]$. Assumption 4.1 implies that $[o] = \text{rel}(o) \forall o \in \text{O}$, and that all generalized objects in $[o]$ correspond to the same instantiated object. Hence, the object link set $\text{OL}$ is also interpreted as a relation over $[\text{O}]$. The elements in $[\text{O}]$ will simply be referred to as objects.

From the base graph six-tuple

$$< \text{O}, \text{PO}, \text{TO}, \text{OL}, \text{PL}, \text{TL}>.$$  

we can form three subgraphs, that each trivially is an ordinary directed graph:

$$< [\text{O}], \text{OL} >, < \text{PO}, \text{PL} > \text{ and } < \text{TO}, \text{TL} >.$$  

(4.3) 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 $< [\text{O}], \text{OL} >$ and the overlaid graph $< \text{PO}, \text{PL} >$. An important property is that even if $< [\text{O}], \text{OL} >$ is not connected, $< \text{PO}, \text{PL} >$ might be. During the package extension described in Section 5.2.2, the pack package links are added only between unconnected parts of $< \text{PO}, \text{PL} >$ and hence $< [\text{O}], \text{OL} >$.

**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., [24].

**Definition 4.5** 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 4.6** Loops and cycles in directed graphs

We define a cycle, in a directed graph as a set of edges, ignoring self-loops, that lead from a node, back to the same node. A directed graph without cycles is called acyclic.

### 4.4.4 Node classifications

In this section, we will introduce some analysis tools for the base graph, for further use in Chapter 5. 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 occurred 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 5.

**Object node classification**

The elements in \([\mathcal{O}]\) are classified according to their local situation in terms of link participation in the base graph, which is expressed as a function:

$$\text{NodeCl}: [\mathcal{O}] \rightarrow 2^{\mathcal{O}}.$$ 

Cf. Figure 4.6(b). The elementary object classifications are

$$\mathcal{C}_o = \{U, I, Ce, Cr, UT, CrT\}.$$ (4.4)

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 4.2. The complete object classification \(\text{NodeCl}(o)\) for an object \(o \in [\mathcal{O}]\) hence consists of any combination of the elementary object classifications, i.e., of any combination of the properties in Table 4.2.
<table>
<thead>
<tr>
<th>$C_O$</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U$</td>
<td>Object has sent a relational unknown message ($rel_u$)</td>
</tr>
<tr>
<td>$I$</td>
<td>Object has sent an internal error message ($int$)</td>
</tr>
<tr>
<td>$Ce$</td>
<td>Object is complainee of a relational link ($rel$ or $rel_{der}$)</td>
</tr>
<tr>
<td>$Cr$</td>
<td>Object is complainer of a relational link ($rel$ or $rel_{der}$)</td>
</tr>
<tr>
<td>$UT$</td>
<td>Object has sent a relational unknown IPC-error message ($rel,T_u^C$ or $rel,T_u^S$)</td>
</tr>
<tr>
<td>$CrT$</td>
<td>Object has sent a relational IPC-error message ($rel,T$)</td>
</tr>
</tbody>
</table>

Table 4.2: Informal explanation of the elementary object classifications in Equation (4.4).

Some examples of how the elementary object classifications are combined are shown in Figure 4.7. The object classification is shown at the lower right corner of the object, and we use the infix operator $\cup$ to denote set union.

The formalization of Table 4.2, i.e., the formal definition of $NodeCl(o)$ for $o \in [O]$, is made with six rules, one for each elementary object classification ($C_O$ in Equation (4.4)).

**Definition 4.7 Object classification**

The *object classification* for a base graph object node in $[O]$ is given by a function

$$NodeCl : [O] \rightarrow 2^{C_O}.$$
defined with six monotone rules:

- **$U$** ∈ NodeCl($o$) iff $\exists o1 ∈ OL$ s.t. cl($o1$) = $rel_u$ and cr($o1$) ∈ $[o]$
- **$I$** ∈ NodeCl($o$) iff $\exists o1 ∈ OL$ s.t. cl($o1$) = $int$ and cr($o1$) ∈ $[o]$
- **$Ce$** ∈ NodeCl($o$) iff $\exists o1 ∈ OL$ s.t. cl($o1$) ∈ \{ $rel$, $rel_{des}$ \} and ce($o1$) ∈ $[o]$
- **$Cr$** ∈ NodeCl($o$) iff $\exists o1 ∈ OL$ s.t. cl($o1$) ∈ \{ $rel$, $rel_{des}$ \} and cr($o1$) ∈ $[o]$
- **$UT$** ∈ NodeCl($o$) iff $\exists tl ∈ TL$ s.t. cl($tl$) ∈ \{ $relT^C_u$, $relT^S_u$ \} and obj($tl$) ∈ $[o]$
- **$CrT$** ∈ NodeCl($o$) iff $\exists tl ∈ TL$ s.t. cl($tl$) = $relT$ and obj($tl$) ∈ $[o]$

An object classification in the base graph hence consists of any element from $2^{CO}$. 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 $|2^{CO}| = 2^6 - 1 = 63$. 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 4.2). The following subset of $2^{CO}$ will be called the *well formed* object classifications:

$$WF_{CO} = \{ U, U|Ce, Ce, I|Ce, Cr, U|Cr, U|Ce|Cr, Ce|Cr, T, Ce|T \}$$

where, e.g., $U$ is interpreted as $\{ U \}$ and the infix operator $|$ is used to denote set union. In Figure 4.8, 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 4.9.

Object classifications from $2^{CO} \setminus WF_{CO}$ 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 4.8: The well formed object classifications $\text{WF}_{C_0}$ from Equation (4.5) (IPC-errors excluded).

by

$$I|Cr, Cr|T, U|I, U|T, I|T \text{ and } UT|CrT$$

(4.6)

and exactly all supersets thereof in $2^{C_0}$, as is easily checked. The classifications in Equation (4.6) will be called the minimal non-well formed object classifications. The graphical notation for the two object classifications in Equation (4.6) not involving IPC-errors are shown in Figure 4.10.

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 4.1.1). 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 (4.6) are the cases where one of the links that should root an error condition propagation chain (classifications $I$, $UT$ and $CrT$) is combined in an object with a link that continues the chain.
Figure 4.9: The well formed object classifications \( \text{WF}_{\text{Cr}} \) from Equation (4.5) involving IPC-errors.

Figure 4.10: The non-well formed object classifications from Equation (4.6) (IPC-errors excluded).

(classifications \( \text{Cr} \) 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 (4.5) 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 (\( \text{rel}_{\text{u}} \)) and a relational error message (\( \text{rel} \)) in the same fault scenario (object classifications \( U|\text{Cr} \) and \( U|\text{Ce}|\text{Cr} \)). The interpretation of these object classifications from a fault isolation point of view is clear for these classifications, though, see Chapter 5. After extension of the base graph using the methods in Chapter 5, objects may get these classifications due to added \( \text{rel}_{\text{der}} \) 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 \( \text{Cr} \) induced by relational (\( \text{rel} \)) and relational derived (\( \text{rel}_{\text{der}} \)) object links (cf. Table 4.2 and Definition 4.7).

If \( U|\text{Cr} \) and \( U|\text{Ce}|\text{Cr} \) are considered non-well formed also, it turns out that the set of classifications whose supersets give all non-well formed classifications from Equation (4.6) is simply extended with \( U|\text{Cr} \). This in turn means that the non-well formed classifications would be characterized exactly as any combination
Table 4.3: Informal explanation of the elementary package classifications in Equation (4.7).

<table>
<thead>
<tr>
<th>( C_P )</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( U )</td>
<td>Object in package has sent a relational unknown error message</td>
</tr>
<tr>
<td>( I )</td>
<td>Object in package has sent an internal error message</td>
</tr>
<tr>
<td>( Ce )</td>
<td>Package is complainee of a package link</td>
</tr>
<tr>
<td>( Cr )</td>
<td>Package is complainer of a package link</td>
</tr>
<tr>
<td>( UT )</td>
<td>Object in package has sent a relational unknown IPC-error message</td>
</tr>
<tr>
<td>( CrT )</td>
<td>Object in package has sent a relational IPC-error message</td>
</tr>
<tr>
<td>( \emptyset )</td>
<td>None of the above</td>
</tr>
</tbody>
</table>

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 \( P_O \) and package links \( PL \), belongs to the object part of the base graph, as discussed in Section 4.4.3. Packages in the base graph are, analogous to objects, classified according to their local situation in terms of link participation, expressed as the function:

\[
\text{NodeCl} : P_O \rightarrow 2^{C_P}
\]

where the *elementary package classifications* are

\[
C_P = \{ U, \ I, \ Ce, \ Cr, \ UT, \ CrT, \ \emptyset \}.
\] (4.7)

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 4.3. Recall from Section 4.4.3 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 \( \emptyset \). 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. The reason is illustrated in Figure 4.11. The nature of the empty elementary package classification \( \emptyset \) is of course such that it always occurs alone.

63
Figure 4.11: How the empty package classification, $\emptyset$, in Equation (4.7) and Table 4.3 may occur.

The formalization of Table 4.3, i.e., the formal definition of $\text{NodeCl}(p)$ for $p \in P_O$, is made with seven rules, one for each elementary package classification ($CP$ in Equation (4.7)).

**Definition 4.8 Package classification**

The package classification for a base graph package node in $P_O$ is given by a function

$$\text{NodeCl}: [O] \rightarrow 2^{CP}$$

defined with seven rules:

- $U \in \text{NodeCl}(p)$ if $\exists o \in p, U \in \text{NodeCl}(o)$
- $I \in \text{NodeCl}(p)$ if $\exists o \in p, I \in \text{NodeCl}(o)$
- $Ce \in \text{NodeCl}(p)$ if $\exists p_1 \in PL \text{ s.t. } \text{ce}(p_1) = p$
- $Cr \in \text{NodeCl}(p)$ if $\exists p_1 \in PL \text{ s.t. } \text{cr}(p_1) = p$
- $UT \in \text{NodeCl}(p)$ if $\exists tl \in TL \text{ s.t. } \text{cl}(tl) \in \{\text{relT}_u^{C_e}, \text{relT}_u^{S}\} \text{ and } \text{obj}(tl) \in p$
- $CrT \in \text{NodeCl}(p)$ if $\exists tl \in TL \text{ s.t. } \text{cl}(tl) = \text{relT} \text{ and } \text{obj}(tl) \in p$
- $\emptyset \in \text{NodeCl}(p)$ if none of the above apply

Analogous to the object node classifications, the following subset of $2^{CP}$ will be called *well formed* package classifications:

$$WF_{CP} = \{U, U|Ce, Ce, I, I|Ce, Cr, U|Cr, U|Ce|Cr, Ce|Cr, T, Ce|T, U|T, U|Ce|T, U|I, U|I|Ce, \emptyset\}$$ (4.8)

where the symbol $T$ us used as a “wildcard” that can be substituted for $UT$, $CrT$ or $UT|CrT$. Package classifications from $2^{CP} \setminus WF_{CP}$ 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 $WF_{CO}$ in Equation (4.5). 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 $WF_{CP} \supset WF_{CO}$, 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

$$I|Cr, Cr|T \text{ and } I|T$$

and exactly all supersets thereof in $2^{Cr}$. These classifications will be called the \textit{minimal non-well formed package classifications}. A direct comparison with the minimal non-well formed object classifications in Equation (4.6) shows that a number of minimal non-well formed classifications are simply removed from Equation (4.6) to Equation (4.9) (the different interpretation of $T$ has no influence, since we are interested in the supersets).

The minimal classifications that are removed from Equation (4.6) to Equation (4.9), 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 (4.9), 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 \textit{well formed} if all elements in $O$ and $P_O$ have well formed classifications.

\textbf{Thread node classification}

The thread part of the base graph, i.e., $<T_C, TL>$, 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 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
Table 4.4: Informal explanation of the elementary thread node classifications in Equation (4.10).

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

\[ C_T = \{ U^C_T, U^S_T, C^T, C^R \}. \] (4.10)

and a brief explanation is given in Table 4.4. Analogous to objects and packages, the thread node classification is given by the function

\[ \text{NodeCl} : T_O \rightarrow 2^{C_T}, \]

and the formal definition of NodeCl(t) for a thread node \( t \in T_O \) is given by the rules

\[
egin{align*}
U^C_T & \in \text{NodeCl}(t) \quad \text{iff} \quad \exists t \in T_L \text{ s.t. cl}(t) = \text{rel}T^C_u \text{ and th}(t) = t \\
U^S_T & \in \text{NodeCl}(t) \quad \text{iff} \quad \exists t \in T_L \text{ s.t. cl}(t) = \text{rel}T^S_u \text{ and th}(t) = t \\
C^T & \in \text{NodeCl}(t) \quad \text{iff} \quad \exists t \in T_L \text{ s.t. cl}(t) \in \{ \text{rel}T, \text{rel}T^\text{der} \} \text{ and ce}(t) = t \\
C^R & \in \text{NodeCl}(t) \quad \text{iff} \quad \exists t \in T_L \text{ s.t. cl}(t) \in \{ \text{rel}T, \text{rel}T^\text{der} \} \text{ and cr}(t) = t.
\end{align*}
\] (4.11)

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 \( U^C_T, U^S_T \) and \( C^R \) is considered non-well formed. The well-formed classifications would be

\[ U^C_T, U^S_T, C^T, C^R, U^C_T|C^R, U^S_T|C^R \text{ and } C^T|C^R \text{.} \]

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 5 actually only uses the NodeCl function to check for the existence of (unexplained) thread node classification \( U^S_T \) (see Table 4.4).
Chapter 5

Explaining error messages

In this chapter, we show how the multitude of error messages in a fault scenario caused by error condition propagation, described in Chapter 4, can be explained by an automatic procedure.

In the explanation models in Chapter 4, when, e.g., a hardware fault has occurred, 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 5.3.2.

According to the object explanation models in Section 4.1.2, 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 4.2.1). An error message can hence be considered caused, or explained, by another error message if they involve the same object. For example, in Figure 5.1(a), the first relational error message (denoted 1) is explained by the second (denoted 2), since they can be assumed to be part of the same error condition propagation chain (see Section 4.1.1). 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 5.1(b).

If the complete error condition propagation that has occurred in the fault scenario is not represented by the base graph subgraph \(<|\mathcal{O}|, \mathcal{O}\mathcal{L}>>\), i.e., if it is not connected, we can attempt to extend the graph. In Section 5.2.1 we show how the structural system model from Section 2.2 and Chapter 3 can be used to infer probable error condition chains and extend the subgraph \(<|\mathcal{O}|, \mathcal{O}\mathcal{L}>>\).

If the subgraph \(<|\mathcal{O}|, \mathcal{O}\mathcal{L}>>\) is not connected, even after extension, the package subgraph \(<\mathcal{P}_\mathcal{O}, \mathcal{P}\mathcal{L}>>\) may still be connected (see, e.g., Example 4.1). As discussed
in Section 4.1.2, we can then assume that this graph represents the error condition propagation that has occurred, 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 \(< P_O, P_L >\) in turn is not connected, it can be extended similar to the object subgraph, as is shown in Section 5.2.2.

According to the discussion in Section 4.1.3, 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 5.2.3, the thread part of the base graph, subgraph \(< T_O, T_L >\), is only considered a complement to the object part, needed when thread communication problems are part of the fault scenario, as discussed in Section 4.1.3. In the current implementation, we make no attempt to connect the thread subgraph. 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 5.2.3.

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 5.1.

In Section 5.2, we show how the structural system model from Section 2.2 and Chapter 3 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 5.3. We will also show that given that the initial base graph fulfills certain properties, the relation over
5.1 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 4 are valid. The error condition propagation must also be traceable, i.e., the propagation must have occurred in the software, and sufficient (relational) error messages must have been sent to the log. Fault propagation occurring via hardware dependencies, i.e., in the environment of the system (see Section 1.4), are not recognized in the current approach. In Example 6.4, we demonstrate a fault scenario where fault propagation has occurred 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 5.3.2 and Examples 6.4 and 6.2.

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 cycles, i.e., two or more system entities (objects or threads) “blame” each other, forming a cycle in one of the base graph subgraphs (see Definition 4.6 and Section 4.4.3). The presence of cycles 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 5.3, cycles in the base graph lead to cycles 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 5.2, is that the initial base graph is acyclic.

Assumption 5.1 (Initial base graph acyclic) In an initial base graph, i.e., a base graph constructed solely from error messages,

\[ \langle |O|, P_O, T_O, OL, PL, TL \rangle, \]

the directed subgraphs

\[ \langle |O|, OL \rangle, \langle P_O, PL \rangle \text{ and } \langle T_O, TL \rangle \]
are acyclic (Definition 4.6). If this property holds, the base graph is called acyclic.

Cf. Definition 4.6. 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 cycles. This will be treated in detail in in Section 5.3.3.

There are 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 cycles, the fault isolation scheme still is a useful tool for system designers and implementors to analyze the fault scenario. Also, if the cycle(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 5.3.

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 6.1). 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.

5.2 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 3 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 $< [O], OL >$ might not be connected. A common situation, already exemplified in Section 1.3 and Example 4.1, 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 6.1.
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 merge of information is treated in Section 5.3. See also Example 6.2.

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 \(< [O], OL >\), is treated in Section 5.2.1. Extension on the package level, i.e., for the base graph subgraph \(< PO, PL >\) is treated in Section 5.2.2.

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 5.2.3 makes no attempt to make the thread part of the base graph, i.e., the subgraph \(< TO, TL >\), connected, and is only meant to explain the relational unknown IPC-error messages.

### 5.2.1 Class extend graph algorithm

In this section, we will show how the system model can be used to make the base graph subgraph \(< [O], OL >\) connected, or at least more connected than before. To this end, we identify the objects in \(< [O], OL >\) 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 \(WF_{CO}\) from Equation (4.5) in Section 4.4.4 are partitioned into two sets:

\[
\begin{align*}
\text{NC}_{CO} &= \{ U, U|Ce, Ce \} \\
\text{OK}_{CO} &= \{ I, I|Ce, Cr, U|Cr, U|Ce|Cr, Ce|Cr, \\
& \quad UT, Ce|UT, CrT, Ce|CrT \}
\end{align*}
\]

Intuitively, objects with classification from \(\text{NC}_{CO}\), 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 \(\text{NC}_{CO}\) are visualized in the leftmost column of Figure 4.8.

### 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,

\[[c_0, \ldots, c_n] \in SC\]

where \(c_1, \ldots, c_n \in \text{Classes}\) and \(n \geq 1\). \(SC\) is the set of all server chains. We also define the selector function

\[\text{last}([c_0, \ldots, c_n]) = c_n,\]
For \( k > 1 \):

\[
\begin{align*}
sc(c, 0) &= \{ [c_0] \mid c_0 \to c \text{ or } c_0 = c \} \\
sc(c, 1) &= \{ [c_0, c_1] \mid c_0 \in sc(c, 0) \text{ and } c_0 \xrightarrow{\sigma} c_1 \in A, s \not\in TS \} \\
sr(c, 1) &= \{ [c_0, c_1] \mid c_0 = c, c \to c \text{ and } c \xrightarrow{\sigma} c_1 \in A, s \not\in TS \} \\
sd_d(c, 1) &= \{ [c_0, c_1] \mid [c_0, c] \in sr(c, 1) \cup sc(c, 1) \text{ and } c_1 \to c \} \\
sd(c, 1) &= sc(c, 1) \cup sr(c, 1) \cup sd_d(c, 1)
\end{align*}
\]

Table 5.1: The server chain function definition.

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

\[
sc : \text{Classes} \times \mathbb{N} \rightarrow \mathcal{SC},
\]

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, \( <\text{Classes}, A, \to> \) as defined in Chapter 3.

As mentioned in Section 3.1, the class associations in the system model may be stereotyped (\( S_A \)). 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

\[
\text{TS} := \{ <<\text{Enqueue}>>, <<\text{Subscribe}>> \}.
\]

As described in Section 2.2.1, 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 5.1. The server chain function is designed to explore all associations in the system model through which a class may depend on another class. In Algorithm 5.1, 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 5.2, where the result of our
actual implementation is shown. The numbering by the classes in Figure 5.2(a) and in the leftmost column of the table in Figure 5.2(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 \( sc(c, 0) \) in Table 5.1 includes all subclasses of the original search class. In \( sc(c, 1) \), all associations, i.e., servers, of these classes are enumerated. \( sc(c, 1) \) 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 \( sc(c, 1) \) have the original search class as first element, even though the association was found in a superclass.

The function \( sc_d(c, 1) \) adds a server chain for all subclasses of the servers found, so that the extension algorithm also matches these against generalized objects in the base graph. As can be seen in Figure 5.2, our implementation does not calculate the parts of \( sc \) as defined in Table 5.1 in order, but the server chains given by \( sc_d \) are enumerated immediately after the corresponding chain given by \( sc \) or \( sc_r \).

Following the rule mentioned above, that class information should be as specific as possible, superclasses of the servers found by \( sc(c, 1) \) and \( sc_d(c, 1) \) are not returned. The extension algorithm (Algorithm 5.1 below) explicitly searches 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 5.1. The definition of \( sc(c, k) \) for \( k > 1 \) is a direct generalization of \( sc(c, 1) \).

**The extension algorithm**

To define the class extension algorithm, we need a function that indicates whether two objects in \(<[O], OL>\) belong to the same connected component (Definition 4.5):

\[
\text{connected} : [O] \times [O] \rightarrow \mathbb{B}.
\]

Simple algorithms for this purpose are well-described in the literature, see, e.g., [24]. Our proposed class extension algorithm is given in Algorithm 5.1.

**Algorithm 5.1 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\(<[O, OL]>,< Classes, \rightarrow, A >\>

**Input:** Base graph, System model

**Output:** Extended base graph \(<[O, 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 5.3(a), and the pseudo-code for the subroutine is given in Figure 5.3(b).
(a) Part of the \(<\text{Classes}, A, -\rightarrow\>\) 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 \(\langle\text{search depth}, <\text{nr}\rangle\) by our implementation of sc.

\[
\begin{array}{c|c|c}
<dpth>, <nr> & \text{Server chain} & \text{Member of} \\
\hline
1.1 & A - C1 & sc(A,1) \\
1.2 & A - B & sc(A,1) \\
1.3 & A - Bsub1 & scd(A,1) \\
1.4 & A - Bsub2 & sc(A,1) \\
1.5 & A - Bsub3 & sc(A,1) \\
1.6 & A - C2 & sc(A,1) \\
1.7 & Asub1 - C3 & sc(A,1) \\
1.8 & Asub2 - E & sc(A,1) \\
2.1 & A - C1 - E & sc(A,2) \\
2.2 & A - B - D1 & sc(A,2) \\
2.3 & A - B - D2 & sc(A,2) \\
2.4 & A - Bsub2 - D3 & sc(A,2) \\
\end{array}
\]

(b) The possible complainees for the original search class A, in the order given by our implementation of the server chain function sc.

Figure 5.2: Example of the server chain function sc defined in Table 5.1.
for the most specific representative of each equivalence class \( o \in \mathcal{O} \) do

if \( \text{cl}(o) \not\in \text{WF}_{\mathcal{C}_0} \) then

issue warning;

elsif \( \text{cl}(o) \in \text{NC}_{\mathcal{C}_0} \) then

\text{classExtendGraph}(o);

end if

end for

(a) The main procedure.

classExtendGraph\( (o \in \mathcal{O}) \)

for \( k = 1 : \text{max}\_\text{depth} \) do

for \( s \in \text{sc}([\text{class}(o); k]) \) do

if \( \exists o' \in \mathcal{O}, \text{class}(o') = \text{last}(s) \) 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

if \( s \not\in \text{sc}_d([\text{class}(o), k]) \) then

if \( \exists o' \in \mathcal{O}, \text{class}(o') \in \text{asc}([\text{last}(s)]) \) 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 5.3: The Class Extend Graph algorithm, Algorithm 5.1.
The algorithm searches the system model for possible complainees (lines 2 and 3 in the subroutine, Figure 5.3(b)) for all generalized objects in $O$ that need a complainee (lines 1 and 4 in the main procedure) according to Equation (5.1).

Only well formed objects in $\{O\}$ are considered, line 2. A non-well formed object is interpreted as a design fault, or that the message log cluster, discussed in Section 4.3, 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 server found, lines 9-10. The last part of the conditions on lines 4 and 10 prohibits the algorithm from adding new cycles to the base graph.

If a complainee candidate already present in $O$ is found, (lines 4 and 10 in Figure 5.3(b)) the corresponding extra generalized objects and object links are added to $O$ and $OL$. Here we have 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 5.2.2. The reason for this choice is simplicity, and that the system model $<\text{Classes}, \text{A}, \to>$ 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 demonstrated to be suitable for the ABB Robotics application, though, since there is usually 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 $o, o' \in O$ with class($o$) $\rightarrow$ 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 $\text{max\_depth}$ on line 2 in Figure 5.3(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 $\text{max\_depth} = 2$ to be a reasonable choice.

Note also that $\text{classExtendGraph}$, if successful, adds a $Cr$ to the classification of the argument object, and hence moves it from $\text{NC}_{Co}$ to $\text{OK}_{Co}$ (Equation (5.1)). 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 $\text{OK}_{Co}$.

### 5.2.2 Package extend graph algorithm

In this section, we show how the system model from Chapter 3, specifically $<\text{PC}, \text{PD}>$, can be used to make the package subgraph of the base graph, $<\text{PO}, \text{PL}>$, 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 \(WF_{CP}\) from Equation (4.8) are partitioned into two parts:

\[
\begin{align*}
NC_{CP} &= \{ U, U|Ce, Ce, 0 \} \\
OK_{CP} &= \{ I, I|Ce, Cr, U|Cr, U|Ce|Cr, Ce|Cr, \\
& \quad T, Ce|T, U|T, U|Ce|T, U|I, U|I|Ce \} \\
\end{align*}
\]  

(5.2)

where packages with classification from \(NC_{CP}\) needs a complainee.

As for the class extension algorithm, we need a graph theory function that indicates whether two packages in the directed graph \(<P_O, PL>\) belong to the same connected component (Definition 4.5):

\[
\text{connected } : P_O \times P_O \rightarrow \mathbb{B}.
\]

Simple algorithms for this purpose are well-described in the literature, see, e.g., [24]. Our proposed package extension algorithm is given in Algorithm 5.2.

**Algorithm 5.2 PackageExtendGraph**

The algorithm searches the system model for possible complainees for packages that needs it, and extends the base graph accordingly.

**Syntax:** PackageExtendGraph\(<P_O, PL>, <P_C, PD>\)

**Input:** Base graph, System model

**Output:** Extended base graph \(<P_O, PL>\)

Pseudo code for the algorithm is given in Figure 5.4.

The algorithm loops through all packages in \(P_O\). For non-well formed packages, a warning is given and that package is ignored (line 2).

For each base graph package \(p_o\) with classification from the set \(NC_{CP}\), Equation (5.2), we find all other packages that have manifested themselves in the fault scenario and are related to \(p_o\) through the package dependency relation \(PD\) in the system model (the set \(P_O\) at line 5).

The package part of the system model, \(<P_C, PD>\), 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 5.1. 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 \(<P_O, PL>\) for all the packages in \(P_O\). 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\((p_o, \hat{p}_o)\), prohibits the algorithm from adding package links that could create cycles 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 \(OK_{CP}\) in Equation (5.2) (lines 6 and 8). Recall from Section 2.2.3 and Section 3.1 that
for $p_o \in P_O$ do
  if $cl(p_o) \notin WF_{C_P}$ then
    issue warning;
  elseif $cl(p_o) \in NC_{C_P}$ then
    $P_O = \{ \hat{p}_o \in P_O : pmap(p_o) \rightarrow pmap(\hat{p}_o) \in PD \}$
    if for any $\hat{p}_o \in P_O$, $cl(\hat{p}_o) \in OK_{C_P}$ and (pmap($\hat{p}_o$) non-global) and
      not connected($p_o, \hat{p}_o$) then
      Add $p_o \rightarrow \hat{p}_o$ to PL;
    elsif for any $\hat{p}_o \in P_O$, $cl(\hat{p}_o) \in OK_{C_P}$ and (pmap($\hat{p}_o$) global) and
      not connected($p_o, \hat{p}_o$) then
      Add $p_o \rightarrow \hat{p}_o$ to PL;
    elseif for any $\hat{p}_o \in P_O$, $cl(\hat{p}_o) \in NC_{C_P}$ and (pmap($\hat{p}_o$) non-global) and
      not connected($p_o, \hat{p}_o$) then
      Add $p_o \rightarrow \hat{p}_o$ to PL;
    elseif for any $\hat{p}_o \in P_O$, $cl(\hat{p}_o) \in NC_{C_P}$ and (pmap($\hat{p}_o$) global) and
      not connected($p_o, \hat{p}_o$) then
      Add $p_o \rightarrow \hat{p}_o$ to PL;
    end if
  end if
end for

Figure 5.4: The Package Extend Graph algorithm, Algorithm 5.2.

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 $OK_{C_P}$ was found, we
turn to packages that themselves need a complainee, i.e., with classification from
the set $NC_{C_P}$. 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 5.2 and Algorithm 5.1, 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 4.4.3). See Example 6.3 for a real fault scenario where the class
extend algorithm failed, and the package extend algorithm was needed.
5.2.3 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 (\(<T_C, TD>\)).

For the ABB Robotics application, we have actually only found it necessary to consider the thread collaborations according to the Enqueue-pattern (see Section 2.2), when the provider thread (the client in the system model) does not provide the consumer thread (the server) with enough data. For modularity reasons, the consumer does not know the identity of 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 \(t \in T_O\) is considered in need of a complainee thread only if

\[
UT^C \in \text{NodeCl}(t) \text{ and } CrT \notin \text{NodeCl}(t),
\]

where the symbols \(UT^C\) and \(CrT\) are defined in Table 4.4 and Equation (4.11) in Section 4.4.4. Furthermore, the thread dependency found in the system model must have the relevant stereotype, in our case \(<\text{Enqueue}>\).

The proposed thread extension algorithm is given in Algorithm 5.3.

**Algorithm 5.3 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(\(<T_O, TL>, <T_C, TD>\))

**Input:** Base graph, System model

**Output:** Extended base graph (\(<T_O, TL>\))

Pseudo code for the algorithm is given in Figure 5.5.

The algorithm loops through all thread nodes \(T_O\) in the base graph and picks out the thread nodes fulfilling the condition in Equation (5.3) (lines 1 and 2). For each such thread node, we find all possible complainees in the system model (\(T_{dep}\) at line 3).

The extension is then attempted in two steps; first the possible complainees from the system model (\(T_{dep}\)) 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 4.4.3). 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 “−”.

79
for all $t_o \in T_O$ do
  if $UT^C \not\in \text{NodeCl}(t_o)$ and $CrT \not\in \text{NodeCl}(t_o)$ then
    $T_{dep} = \{td \in TD \mid \text{server}(td) = \text{tmap}(t_o) \text{ and}
    \text{stereotype}(td) = \langle\langle Enqueue\rangle\rangle\}$
  for $td \in T_{dep}$ do
    if $\exists \tilde{t}_o \in T_O$ s.t. $\text{tmap}(\tilde{t}_o) = \text{client}(td)$ then
      add $tl = t_o \xrightarrow{relT_{dep}} \tilde{t}_o$ to $TL$
    end if
  end for
  if $\text{client}(T_{dep}) \cap \text{tmap}(T_O)$ is empty then
    for $td \in T_{dep}$ do
      add a thread node $\tilde{t}_o$ with $\text{tmap}(\tilde{t}_o) = \text{client}(td)$ to $T_O$
      add $tl = t_o \xrightarrow{relT_{dep}} \tilde{t}_o$ to $TL$
    end for
  end if
end if
end for

Figure 5.5: The Thread Extend Graph algorithm, Algorithm 5.3.

The second attempt at extending for a thread node fulfilling the conditions in Equation (5.3), is given by the lines 9–15. Since a thread with problems often is silent, see Section 4.1.3, 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 for this, 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 $\langle\langle Enqueue\rangle\rangle$. An example with the thread extension algorithm is given in Example 6.3.

5.3 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.
5.3.1 A relation over the error messages

The cause-effect relation will be defined over the set of links

\[ L = \text{OL} \cup \text{TL} \cup \text{TL}_e \]

where \( \text{TL} \) and \( \text{OL} \), defined in Section 4.4.3, 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 5.2. The here introduced link set \( \text{TL}_e \) 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

\[ l_1 \xrightarrow{k} l_2 \in R \subseteq L \times K \times L. \]

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,

\[ K = \{1, 2\}, \]

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

\[ \{ (l_1, l_2) | \exists k \text{ s.t. } l_1 \xrightarrow{k} l_2 \in R \} \]

is the actual (binary) cause-effect relation over the error messages. The intuitive interpretation of an element \( l_1 \xrightarrow{k} l_2 \in R \), 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 4.1. 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 4.4.4). \( R \) 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 4.1 (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 6.2 and 6.3). 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 6.1, Figure 6.2).

Notation used in this section  We introduce some standard notation for use in this section. In what follows, ol₁, ol₂ and ol₃ are object links where ol₁ ≠ ol₂; tl₁, tl₂ and tl₃ are thread links where tl₁ ≠ tl₂; l ∈ L is an arbitrary link, and p is a package. In the figures, we also use superscripts, e.g., ol₁¹, ol₂¹, to denote different links. We define the two auxiliary functions

\[ nCe: OL \rightarrow [O] \]
\[ sender: L \rightarrow [O] \]

as

\[ nCe(ol₁) = \begin{cases} [cc(ol₁)] & \text{if } cl(ol₁) \in \{ rel, rel_{der} \} \\ [cr(ol₁)] & \text{if } cl(ol₁) = rel_u \\ [cr(ol₁)] & \text{if } cl(ol₁) = int \\ undef & \text{if } cl(ol₁) = undef \end{cases} \]
\[ sender(l) = \begin{cases} [obj(l)] & \text{if } l \in TL \text{ and } cl(l) \neq relT_{der} \\ undef & \text{if } l \in TL \text{ and } cl(l) = relT_{der} \end{cases} \]

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 o₁ complains on another object o₂ which in turn has made a further complaint, we assume that an object-to-object error condition propagation, as described in Section 4.1, has occurred from o₂ to o₁. Hence the first error message is explained by the second one. Rule 1a explains object-to-object fault propagation using the < [O], 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: \( nCe(o_1) = [cr(o_2)] \) and \([cl(o_1), cl(o_2)] \neq [rel_a, rel_a] \Rightarrow o_1 \rightarrow o_2 \in R \)

Rule 1b: \( nCe(o) = [obj(tl)] \Rightarrow o_1 \rightarrow tl \in R \)

(a) The base graph situations. (b) The explanation graph

Figure 5.6: Three base graph situations \( (<[O], OL>) \) covered by Rule 1a. The corresponding explanation graphs are the same for all three situations.

Interior package explanations

Sometimes the occurred fault propagation is not represented in the graph \( <[O], OL> \), even after the class extend graph algorithm in Section 5.2.1 has been applied. We

(a) The base graph. (b) The explanation graph

Figure 5.7: The base graph situations \( (<[O], OL>) \) covered by Rule 1b.
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 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:** nCe(o₁₁) ∈ p, cr(o₁₂) ∈ p and cl(o₁₂) = int → o₁_\text{\textup{\textsuperscript{1}}} \rightarrow o₁_\text{\textup{\textsuperscript{2}}} ∈ R

**Rule 2b:** nCe(o₁) ∈ p, obj(t₁) ∈ p → o₁ \rightarrow t₁ ∈ R

The base graph situations that these rules cover are exemplified in Figure 5.8 and 5.9.

**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.
Figure 5.9: Examples of base graph situations \(<O_P,OL>) covered by Rule 2b.

Rule 3a: \(nCe(ol_1) \in p_1, cr(ol_2) \in p_1, ce(ol_2) \in p_2 \text{ and } p_1 \neq p_2 \implies ol_1 \rightarrow_1 ol_2 \in R\)

Rule 3b: \(nCe(ol) \in p_1, p_1 \stackrel{\text{pack}}{\rightarrow} p_2 \in PL\) and \(\text{sender}(l) \in p_2 \implies ol \rightarrow_1 l \in R\)

In Rule 3a, the object link \(ol_2\) induces a package link \(pl = p_1 \rightarrow_1 p_2 \in PL\). The base graph situations that these rules cover are exemplified in Figure 5.10 and 5.11. In Figure 5.11(a), * is used to symbolize any of the object or thread link classifications \(int, rel_u\) and \(relT^*_a\). The non-well formed cases (see Section 4.4.4) for the second package must of course be omitted.

**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 \((relT_{der})\) from the same thread, added by the thread extension algorithm described in Section 5.2.3, the derived thread link strongly explains the other (Rule 4b). Finally, a chain of thread messages indicates causality (Rule 4c).

Rule 4a: \(cl(tl) \in \{relT, relT_{der}\} \text{ and } th(ol) = ce(tl) \implies tl \rightarrow_1 ol \in R\)

Rule 4b: \(cl(tl_1) \in \{relT^C_u, relT^S_u\}, cl(tl_2) = relT_{der} \text{ and } cr(tl_1) = cr(tl_2) \implies tl_1 \rightarrow_2 tl_2 \in R\)

85
Rule 4c: $cl(tl_1) \in \{ relT, relT_{der} \}$ and $ce(tl_1) = cr(tl_2) \implies tl_1 \xrightarrow{} tl_2 \in R$

The base graph situations covered by these rules are exemplified in Figure 5.12, 5.13 and 5.14. For Rule 4c, Figure 5.14(a), it should be noted that $tl_2$ can also be a relational unknown IPC-error message.

As pointed out in Section 4.1.3, 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 $TL_e$, 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 6.3).

To formulate the rule, we formally call a thread $t \in T_O$ silent if

$$\{ tl \in TL \mid cr(tl) = t \} = \emptyset \text{ and } \{ ol \in OL \mid th(ol) = t \} = \emptyset.$$ 

We also introduce the function $complaints : T_O \rightarrow 2^{TL}$

$$complaints(t) = \{ tl \in TL \mid ce(tl) = t \}.$$ 

The rule then becomes
Figure 5.11: Examples of base graph situations \((< O, P_O, O_L, P_L>)\) covered by Rule 3b.

**Rule 5:** \(t \in T_O\) silent, and \(|\text{complaints}(t)| > 1 \implies\)

\[
\text{add } tl_e = t_{intT_{der}}^\rightarrow t \text{ to } TL_e, \text{ and } tl \rightarrow tl_e \in \mathbb{R} \forall tl \in \text{complaints}(t)
\]

See Example 6.2. The link classification \(intT_{der}\) was introduced in Equation (4.2), Section 4.4.3.

### 5.3.2 Interpretation of the explanation relation

The explanation relation, base graph and the fault isolation scheme as a whole has at least two different usages:

- To be a part of the automatic fault isolation layer discussed in Chapter 1.
- 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 4.4.1 and Section 5.3 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.

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 so far, the answer would be that the fault scenario is well recreated, given that the assumptions discussed in Section 5.1 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 4.5)
- Unique maximal element
Acyclic (Definition 4.6)

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 \( \text{<} P_O, PL \text{>} \) is connected. If in addition \( \text{<} O, OL \text{>} \) 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.

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 occurred, or bad clustering of log.
- The error condition propagation is not traceable.
- No error message directly corresponds to the occurred fault, or fault propagation in the environment (see Example 6.4).

As already mentioned in Section 5.1, loops in the base graph subgraphs indicate an unclear internal error handling in the system. Recall from Section 5.1, Assumption 5.1, 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 5.1), 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

Figure 5.13: Examples of base graph situations (\( < T_O, TL > \)) covered by Rule 4b.
thread part and object part of the base graph (we have yet to encounter examples of this, though). In Section 5.3.3 below, we will present sufficient properties that guarantee that $R$ is acyclic. They can be taken as a design guideline for the control system.

### 5.3.3 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, $\bar{R}$.

\[
\begin{align*}
1_1 & \xrightarrow{k_1} 1_2 \in R & \Rightarrow & & 1_1 \xrightarrow{k} 1_2 \in \bar{R} \\
1_1 & \xrightarrow{k_1} 1' \in R \text{ and } 1' & \xrightarrow{k_2} 1_2 \in R & \Rightarrow & & 1_1 \xrightarrow{\min(k_1,k_2)} 1_2 \in \bar{R} \\
1 & \in L & \Rightarrow & & 1 \xrightarrow{2} 1 \in \bar{R}
\end{align*}
\]

The transitive closure is taken by stating that if there is an explanation path from message $1_1$ to message $1_2$, 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 $\bar{R}$ 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 5.1, 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 5.1** $R$ partial order

If the base graph is well-formed and the base graph subgraphs $< [O], OL >$ and
<P_O, PL > are acyclic, and TL is empty, then R is anti-symmetric and hence a partial order.

Trivially, R is anti-symmetric iff R is acyclic. The proof is mostly a matter of pointing out some facts.

Proof (Theorem 5.1) 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 R anti-symmetric.

We will prove the claim by exhaustively applying the rules to construct an originally empty relation 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 [O] to add an edge between two object links in OL to 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 complainee. Note that cl(o11) ≠ int due to the definition of nCe. The rule trivially does not introduce loops in R for the links involved in the same object.

Since the rule for a specific object in [O] only adds edges to R from incoming links to non-incoming links, and no two objects in the directed graph < [O], OL > are connected, no cycles in R are introduced by Rule 1a (it can be noted that the above argument would cover also Rule 1b).

Rule 2a: For each package \p in P that contains an object, o ∈ [O], that has sent an int error message, the rule introduces an edge in 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 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 5.8).

Rule 3a and 3b: Both rules introduce edges in R according to the base graph subgraph < P_O, PL >. Rule 3b introduces explanations in R from certain object links in a package \p in P that is complainer of a package link \p pack \p_2 ∈ PL, to all links sent by objects in \p. The rule does not introduce cycles by itself since < P_O, PL > is acyclic. The only way for cycles to be introduced in the hitherto 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 \textit{obj} links in \( \mathcal{PL} \), any such cycle would be present also in \( < \mathcal{PO}, \mathcal{PL} > \), though.

Rule 3a introduces explanations in \( \tilde{R} \) from certain object links in a package \( p_1 \in \mathcal{PO} \) that is complainer of a package link \( p_1 \xrightarrow{\text{obj}} p_2 \in \mathcal{PL} \), to the object link(s) that has (have) induced the package link. The “certain object links” in package \( p_1 \) are relational unknown object links sent by objects in \( p_1 \) or relational object links with their complainee in \( p_1 \). An \textit{int} (or \textit{relT}) in \( p_1 \) would not be well-formed. The rule do not introduce cycles in \( \tilde{R} \) by itself or in combination with Rule 3b and Rule 1a, since \( < \mathcal{PO}, \mathcal{PL} > \) is acyclic.

Since the above rules are the only applicable under the given conditions, we have \( \tilde{R} = R \), and the claim is proved. \( \blacksquare \)

The reason we must exclude IPC-error messages for the theorem, is that the base graph object part, \( < [O], \mathcal{OL} > \) and \( < \mathcal{PO}, \mathcal{PL} > \), might overlap in an unfortunate way with the thread part, \( < \mathcal{T}, \mathcal{T}L > \). 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 \( \tilde{R} \) is a partial order under Assumption 5.1.

In Example 6.2 and Example 6.3, 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 6

Implementation and examples

The fault isolation scheme presented here has been implemented and tested on a large scale industrial control system developed by ABB Robotics.

6.1 DrRobot

The fault isolation tool, with the working name DrRobot, is a prototype implementation of the fault isolation scheme presented here. The tool attempts 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\textsuperscript{tm}.

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 4.2.1, 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, were acquired by studying the actual control system source code and from interviews with the system developers at ABB Robotics. It should 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\textsuperscript{tm}, see Section 2.2. DrRobot accesses the system model using the Automation interface, Rose Extensibility Interface, supplied with Rational Rose\textsuperscript{tm}.

\(^1\)See [11] for a thorough discussion on how to count lines of code.
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 3 for suitable data structures.

To create the internal representation of the initial base graph for a specific fault scenario, see Figure 4.6 in Section 4.4.2, 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 4.4.1 are visualized. The explanation graph is constructed from the current base graph as described in Section 5.3 and also visualized. See Section 6.2 for several examples. The user applies the extension algorithms presented in Section 5.2 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 5.3, 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 5.3.1 is in the current implementation given the ID number -42, to distinguish it from the derived links added to the base graph (see Example 6.2). The different trust levels of the explanations in the explanation graph are visualized by making the weaker explanation arrows gray.

6.2 Examples

The ABB Robotics industrial robot control system was presented in Section 2.3, and consists of approximately 2 million lines of code and ca. 500 classes. The system model, which does not describe 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 5.1 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 occurred 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 occurred 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 6.1 Field bus fault scenario, cont.
The example is a continuation of Example 4.1 in Section 4.4.1, where the ABB robot control system is running a user-defined program, when a field-bus unit suffers a fault.

The Figure 6.1 shows the initial base graph, already shown in Figure 4.3, and the corresponding explanation graph constructed with the rules in Section 5.3.1. Already at this point, 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 5.3.1. 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 5.1 (ClassExtendGraph) the base graph, suitable associations are found in the system model and derived links and an in-between generalized object are added to <O, OL>, as shown in Figure 6.2. The conclusion before and after the extension is basically the same, the error message 71107 is closest to the actual fault, but Algorithm 5.1 strengthens the result somewhat. After the extension, <O, 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 6.2 IPC fault scenario, cont.
The example is a continuation of Example 4.2 in Section 4.4.1. Here we use the visualizations of the different graphs produced by the fault isolation tool, instead of the edited versions used in Example 4.2.

We repeat what happened in the fault scenario from Example 4.2. The thread named 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 6.3.

The initial base graph and explanation graph are shown in Figure 6.4. 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 6.4(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.

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 5.1 (ClassExtendGraph) to the initial base graph anyway, the only difference
Figure 6.1: The initial base graph ($< O, P_O, OL, PL >$) and explanation graph from the fault scenario in Example 6.1.
Figure 6.2: The extended base graph (<O, OL>) and explanation graph from the fault scenario in Example 6.1.
is that a relational derived object link (rel\text{der}) is added from the base graph object safevt to safdev in Figure 6.4(a), hence the explanation from 90002 to 90001 in the explanation graph, Figure 6.4(c) is “upgraded” to a strong explanation. The Algorithm 5.2 (PackageExtendGraph) has no effect.

Example 6.3 Real time fault scenario, cont.
The example is a continuation of Example 4.3 in Section 4.4.1. The scenario is that during normal operation of the system, one thread feeds another with data according to the enqueue-pattern (see Section 2.2.1). The fault that occurs 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, servotask 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 6.5. As seen in Figure 6.5, the initial base graph does not contain enough information to recreate the fault scenario especially well. By applying Algorithm 5.2 (PackageExtendGraph) and Algorithm 5.3 (ThreadExtendGraph), we get the extended base graph and corresponding explanation graph shown in Figure 6.6. The Algorithm 5.1 (ClassExtendGraph) has no effect.

The derived thread link in Figure 6.6(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.

Example 6.4 Short-circuit
We present an example of a fault scenario where fault propagation has occurred 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 6.7. As can be seen, the explanation graph gives two maximal error messages; one regarding the bus and
Figure 6.4: The initial base graph and explanation graph from the fault scenario in Example 6.2.
Figure 6.5: The initial base graph and explanation graph for the fault scenario in Example 6.3.
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 5.1 (ClassExtendGraph) connects the base graph better, like in Example 6.1, but the two maximal elements of the explanation graph are the same. The other extension algorithms have no effect.
Figure 6.7: The initial base graph and explanation graph for the fault scenario in Example 6.4.
Chapter 7

Conclusions

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 scenario. 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


Index

A, see system model, association relation
[] , see class equivalence relation
→ , see system model, inheritance and base graph, inheritance
int, rel, relu, relder, obj, pack, relT, relTuC, relTuS, relTder, intTder, see link, classifications
U, I, Ce, Cr, UT, CrT, θ, see base graph, package node classification
U, I, Ce, Cr, UT, CrT, see base graph, object node classification
acyclic, see directed graph and base graph and explanation graph
asc(), 30
association, see link, association
association relation, see system model, association relation
backward relational, see error condition, IPC
base graph, 41, 47
acyclic, 69
connected, 13, 57, 67, 70
extension, 67
generalized object, 44, 47
inheritance, 57
links (OL, PL, TL), 55
nodes (O, Po, TO), 55
object node classification, 58
objects ([O]), 57
package node classification, 63
parts, 57
subgraphs, 57
thread node classification, 65
views, 48
well formed
object classifications, 60
package classifications, 64
class(), 55
cr(), 56
cycle, see directed graph, cycle
desc(), 30
directed graph
acyclic, 58
connected, 57
connected component, 57
cycle, 58
DrRobot, 4, 93

C0L, C0L, C0L, see link, classification
cause-effect relation, see explanation graph
ce(), 56
c1(), see link, classification
c1e(), 31, 33
class, see object, class
class diagram, see UML, class diagram
class equivalence relation, 31, 48
Classes, see system model, Classes
ClassExtendGraph, 73
cluster, see error message, cluster
class(), 55
complainee, 37
complainer, 37
complaints(), 86
component diagram, see UML, component diagram
connected, see directed graph and base graph and explanation graph
connected(, ) 73, 77
cr(), 56

DrRobot, 4, 93
encapsulation, 3, 18
enqueue pattern, see thread, enqueue pattern
error condition
IPC, 38
error condition, 35
  internal, 37
  IPC
    backward relational, 39
    relational, 37
    root, 35
error condition propagation chain, 35
error message, 40
  cluster, 40, 46
  log, 16, 40
  maximal, see explanation graph,
    maximal element
  operational, 45
  run-time
    graphical notation, see base graph
    internal, 41
    IPC, 41
    ordinary, 41
    relational, 41
    signature, 8, 41
    supervisory, 45
error message classification, see error message, run-time
exception handling, 7
explanation graph, 13, 67, 81
  acyclic, 90
  connected, 88
  explanation rules, 82
  maximal element, 4, 67, 87
  trust level, 81
explanation model, 8, 35
  object, 13, 37
  package, 16, 37
  thread, 38
explanation relation, see explanation graph
fault, 6, 34
  primary, 34
  unhandled, 17
  unobservable, 46
fault scenario, 34
generalized object, see base graph, generalized object
inheritance, see object, inheritance
inheritance relation, see system model, inheritance
instance, see object and thread and link
IPC (Inter-Process Communication), 21
IPC-error message, see error message, run-time, IPC
link
  association, 19, 24, 31
  classification, 55
  derived, 13
  instance, 19
maintaining software, 19
maximal error message, see explanation graph, maximal element
modularity, 3, 18
NC_Co, 71
NC_Cp, 77
nCe(), 82
NodeCl(), 58, 63
[O], see base graph, objects
O, see base graph, nodes
OK_Co, 71
OK_Cp, 77
obj(), 56
object, 18
  active, 21, 25
  class, 19
    abstract, 24
    inheritance, 23, 30, 57
    instance, 19, 48
object error message, see error message, run-time, ordinary
object part, see base graph, parts
object view, see base graph, views
OL, see base graph, links
 OMG, see UML

PC, see system model, packages
PO, see base graph, nodes
package, 25, 31
global, 32
package dependency relation, see system model, package dependency relation
PackageExtendGraph, 77
PD, see system model, package dependency relation
PL, see base graph, links
pmap(), 55
primary fault, see fault, primary process, see thread
R, see explanation graph
Rational Rose™, 20, 93
rel(), 30, 57
relation, see explanation graph
sc(), see server chain
sender(), 82
server(), 31, 33
server chain, 71
stereotype, see UML, stereotype
stereotype(), 31, 33
subscription pattern, see thread, subscription pattern
system model
thread dependency relation (PD), 33
system log, see error message, log
system model, 8, 13
association relation (A), 30
class model, 30
Classes, 30, 55
inheritance (→), 30
justification, 16
PC, 55
package dependency relation (PD), 30
packages (PC), 30
TC, 55
thread model, 30
threads (TC), 33

TC, see system model, threads
TO, see base graph, nodes
task, see thread
task diagram, see UML, task diagram
TD, see system model, thread dependency relation
thread, 19
enqueue pattern, 21, 25, 31
instance, 49
silent, 86
starvation, 39
subscription pattern, 21, 25, 31
th(), 56
thread error message, see error message, run-time, IPC
thread part, see base graph, parts
thread view, see base graph, views
ThreadExtendGraph, 80
TL, see base graph, links
TLc, 81
tmap(), 55
trust level, see explanation graph, trust level

UML, 19
class diagram, 20
component diagram, 20
stereotype, 20, 25
task diagram, 21, 27

WFco, see base graph, well formed, object classifications
WFce, see base graph, well formed, package classifications
well formed, see base graph