Fault Isolation in Object Oriented Control Systems

Magnus Larsson*, Inger Klein*, Dan Lawesson** and Ulf Nilsson**

* Department of Electrical Engineering
Linköping University, SE-581 83 Linköping, Sweden
WWW: http://www.control.isy.liu.se
Email: magnusl,inger@isy.liu.se

** Department of Computer and Information Science
Linköping University, SE-581 83 Linköping, Sweden
WWW: http://www.ida.liu.se
Email: danla,ulfni@isy.liu.se

12 December, 2000

REGLERTEKNIK
AUTOMATIC CONTROL
LINKÖPING

Report no.: LiTH-ISY-R-2324
Presented at SAFEPROCESS 2000

FAULT ISOLATION IN OBJECT ORIENTED CONTROL SYSTEMS

Magnus Larsson [*] **Inger Klein** [*] **Dan Lawesson** [**]
Ulf Nilsson [**]

[*] *Dept. of Electrical Engineering, Linköping University, Sweden*
[**] *Dept. of Computer and Info. Science, Linköping University, Sweden*

Abstract: This article addresses the problem of fault propagation between software modules in a large-scale control system with object oriented architecture. There exists a conflict between object-oriented design goals such as encapsulation and modularity, and the possibility to suppress propagating error conditions. The propagation manifests itself as many irrelevant error messages, and hence causes problems for system operators and service personnel when attempting to isolate the real fault. We propose a fault isolation scheme aimed at achieving clear and concise fault information to the operator without violating encapsulation and modularity. The approach is implemented and tested on a commercial industrial robot control system from ABB Robotics and a patent application has been filed with the Swedish patent office (PRV) Larsson and Eriksson (1999).

Keywords: Fault isolation, object modeling techniques, control system, safety-critical, propagation

1. INTRODUCTION

Developing control systems for complex systems is a difficult and increasingly important task. Traditional software development methods based on structured analysis and functional decomposition (see e.g. DeMarco (1979)) are today often replaced by object oriented methods, see e.g., Douglass (1998); Rumbaugh et al. (1991). The new methods have many advantages over traditional approaches, including better possibility to master complexity and to facilitate maintenance and reuse (see e.g. Booch (1994)). However, new problems arise; the problem addressed here is *fault propagation* in an object oriented software architecture for a large-scale, configurable and safety critical control system. As basic inspiration and case study we have used a commercial control system for industrial robots developed by ABB Robotics; the system is highly configurable, programmable and has an object oriented 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 situation, and to avoid propagating error messages. Error messages are sent by individual objects to notify an operator that an error condition has been detected. The aim to encapsulate information implies that individual objects, or groups of objects, in general do not know how close they are to a fault or if the fault has already been adequately reported.

The focus of this paper is an operational and safety critical control system running without direct supervision; 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. Operators or service personnel called in when the system halts due to a failure are fairly unexperienced with the system and have little insight in its internal design. Since error reporting often reflects the internal design of the system, it can be very difficult for the operator to understand which error message that is most relevant and closest to the fault. In this paper we propose a liberal error reporting policy in combination with a fault isolation layer hiding the log and the core control system from the operator; the layer performs post-processing of the fault information and is able to present clear and concise fault information to the operator; thus facilitating design principles such as encapsulation and modularity.

It should be noted that the number of error messages or alarms in a fault scenario need not be especially large to cause problems for an unexperienced operator, the number typically ranges from 3 to 20 in our use case. The strength of the proposed approach does not lie in the number of error messages handled in each fault scenario, but in the wide range of potential fault scenarios handled by a general method.

2. INFORMATION USED DURING FAULT ISOLATION

We propose a fault isolation scheme where error messages are explained locally, by means of information already available to an object at run-time; thus not violating the principle of encapsulation and modularity. This local information, made available in an error message, is called the *error message signature* which together with a conceptual *explanation model* makes it possible to infer cause-effect relations between the error messages. The most relevant error message(s) can then be presented to the operator. When the information from the error messages is inconclusive, we use a structural system model to find dependencies between objects and hence "fill in the gaps". To the authors' knowledge, this is a novel approach.

Error messages are divided into *internal* error messages, and *relational* error messages; in a relational message the complaining object is called the *complainer*, and the imputed object is called the *complainee*. Relational error messages are further specialized into those where (1) the complainee is known and where (2) the complainee is unknown. If the system is regarded as a collection of collaborating, fairly intelligent, but narrow-minded individuals, these three types can be characterized with the statements "I did it", "he did it" and "I didn't do it" respectively.

The information provided in the error messages is complemented with a *structural system model*. A model was needed since the software in itself was far too complex (in the order of 10^6 lines of code in the ABB case). Even if it is not yet common practice to do so, it is widely recognized that developing system models helps in designing correct systems; hence it is not unreasonable to assume that software in the future will be accompanied by models at different levels of abstraction. The modeling language used here was the Unified Modeling Language (UML)(see e.g., Douglass (1998)). The UML is a design notation for object oriented systems and also serves as system documentation. For the fault isolation process we use the UML *class diagrams* and *task diagrams*.

Closely collaborating and related classes can in the UML be collected into modules called *packages*. 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. How this information is used in the fault isolation approach will be demonstrated below.

For the system model (in the form of UML class diagrams) to be useful for fault isolation, the system and system model should be such that the static class structure reflects the run-time object structure well. Classes in control systems are often highly specialized. Even if the complete run-time object structure often is very dynamic and constantly changing, there are usually only a few "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. Another important property is that the inheritance hierarchies are seldom very deep; often only one or two levels.

Since the information used for fault isolation is partitioned into a UML model and error message signatures, the approach scales quite well. The fault isolation scheme is easy to maintain and to extend when the system changes, since it is an integral part of the software development process and the software itself.

3. A FAULT ISOLATION SCENARIO

Due to space limitations it is not possible to fully describe the formal notation or algorithms used in the implemented system. Instead we illustrate the method by a real fault scenario from the ABB Robotics industrial robot application. For a complete, and formal, treatment see the thesis Larsson (1999) or the report Larsson et al. (1999). The purpose of the example is to illustrate the fault isolation scheme, and we will not go into system details.

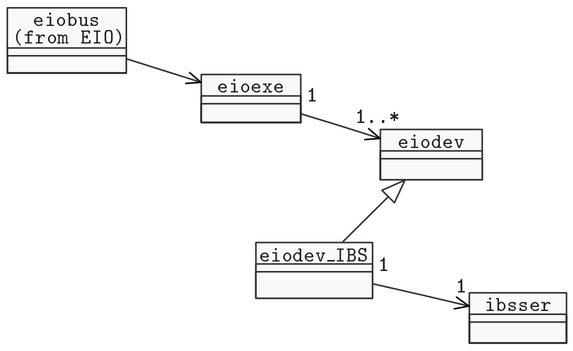
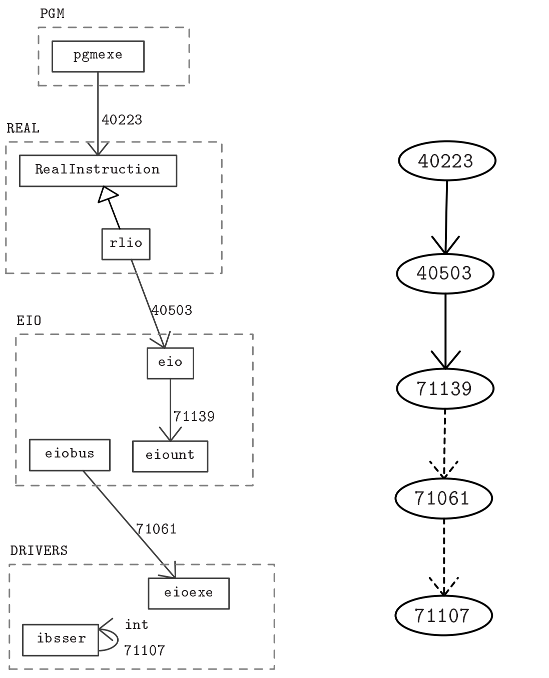


Fig. 1. Extract of class diagram from the package `Drivers`.

```
7.  10008 Program restarted 0105 13:45.9
The task MAIN has
restart to execute.
The originator is the production window.

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

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

10.  71139 Access error from IO 0105 13:45.35
Description\Reason:
- Cannot Read or Write signal DO3_1
  due to communication down.

11.  40503 Reference error 0105 13:45.35
Device descriptor is
not valid for a digital write operation

12.  40223 Execution error 0105 13:45.35
Task MAIN: Fatal runtime
error

13.  10020 Execution error state 0105 13:45.35
The program execution has reached
a spontaneous error state

14.  10005 Program stopped 0105 13:45.35
The task MAIN has
stopped. The reason is that
an external or internal stop has
occurred.
```

Fig. 2. Error log for the example.

In Figure 1, part of the system model relevant to our fault scenario is shown in UML class diagram notation. Classes are shown graphically using rectangles with the name of the class inside. That an object uses some service performed by another object is modeled with an arrow between classes, a so called *association*. A class can be a specialization of a more general class; this is called *inheritance*, and is indicated by an arrow with a triangle head.

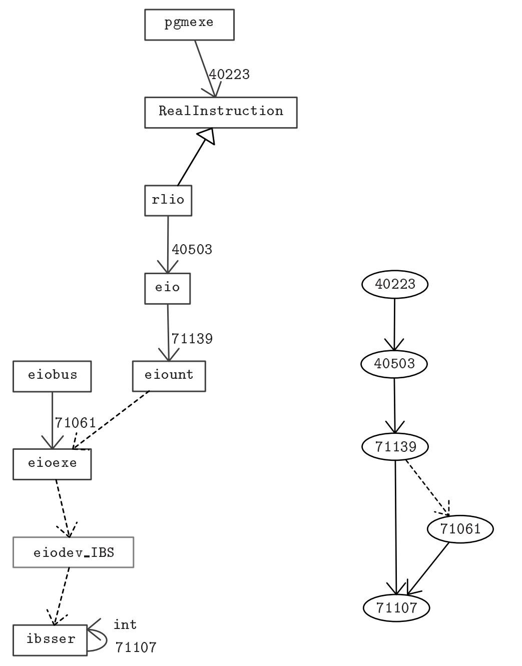


(a) Original base graph. (b) Explanation graph.

Fig. 3. Original base and explanation graphs.

The fault considered here is a malfunctioning field bus. The resulting error message log is given in Figure 2. The error message signatures are not shown in the log, but the local information provided in the error message signatures is visualized in a so-called *base graph*, see Figure 3(a). Each node of the base graph corresponds to an object that has either sent an error message (a complainer) or is pointed out by another object (a complainee). The edges between nodes correspond to relational error messages and should be read "complains on". The self-loop adorned with `int` corresponds to an internal error message. There is also one inheritance relation in the base graph. The packages are shown using dashed boxes with the package name in the upper left corner.

The base graph describes dependencies between objects, but the aim is to point out the error message closest to the fault. For this purpose we construct an explanation graph, see Figure 3(b). The explanation graph is in some sense the dual of the base graph; the nodes correspond to error messages and the edges represent dependencies between error messages. The goal of the fault isolation scheme is to produce a connected explanation graph without any cycles where all error messages can be traced to one primary error message. This error message can then be presented to the operator. In our scenario this primary error message is message number *71107*. Note that the base graph is not connected on the class level,



(a) Extended base graph. (b) Explanation graph.

Fig. 4. Extended base graph and explanation graph.

but by using the package information we achieve a connected explanation graph anyway.

If the base graph is not connected, as in our case, it may be necessary to extend the base graph using the UML system model (Figure 1). This can be done both on the class- and package levels. Algorithms for doing this are further described in Larsson (1999); Larsson et al. (1999). In our example, an extension on the class level is possible. The basic idea is to try to find complainees for objects in the base graph which do not have "somebody to blame".

The system model is searched for associations between classes corresponding to objects in the base graph, possibly via intermediate classes. The result is an extended base graph and corresponding explanation graph as in Figure 4. Note that the conclusions of the fault isolation approach are strengthened by the extension, even though the original explanation graph already was connected.

In this example the generation of the explanation graph is easy, but the situation becomes more complicated if, e.g., IPC error messages are present (communication errors between concurrent tasks). In those cases the base graph consists of two parts, one based on the class diagrams and one based on task diagrams, and the explanation graph is constructed using the whole base graph.

4. CONCLUSIONS

We have presented a scheme for fault isolation in object oriented control systems. The method is based on the error messages in the error log, and uses a UML model of the system to complete the explanation graph which shows the cause-effect relationships between error messages. The strength of the proposed approach does not lie in the amount of error messages handled in each fault scenario, but in the wide range of potential fault scenarios handled by a general method.

The method outlined here has been implemented. The core of the fault isolation layer consists of ca. 2000 lines of C++ code and is able to access UML models developed in Rational Rose. The algorithms have been tried on a set of real fault scenarios from the ABB Robotics industrial robot control system. In the nine examples considered, the system was able to pin-point the primary error message in seven cases – in the remaining two cases the error was caused in a sub-system that was not part of the UML model – hence, there was no hope of pin-pointing the error. In those cases, the fault isolation tool returned several maximal error messages, but offered a deepened insight in the fault scenario.

The system model used above captures the *structure* of the system. In our future work we will examine the possibility to use a system model containing also behavioral information. Naturally, a more detailed model allows for more precise diagnosis, but it also poses problems in terms of maintenance of the model and finding (in some sense) correct rules for reasoning. State charts are included in the UML, and they are our present candidate for a behavior system model. One way of performing reasoning would then be to use a model checker.

References

G. Booch. *Object-Oriented Analysis and Design: With Applications.* Benjamin/Cummings, 2 edition, 1994.

T. DeMarco. *Structured Analysis and System Specification.* Prentice-Hall, 1979.

B. P. Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems.* Addison Wesley, 1998.

M. Larsson. *Behavioral and Structural Model Based Approaches to Discrete Diagnosis.* Phd thesis 608, Department of Electrical Engineering, Linköping University, Linköping, Sweden, 1999. Can be aquired at http://control.isy.liu.se/publications/.

M. Larsson and P. Eriksson. Fault isolation in process. Swedish Patent Application No. 9904008-1, 1999.

M. Larsson, I. Klein, D. Lawesson, and U. Nilsson. Model based fault isolation for object-oriented control systems. Technical Report LiTH-ISY-R-2205, Dept. of Electrical Engineering, Linköping University, 1999. Can be acquired at `http://control.isy.liu.se/publications/`.

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.