A Meta-Model for Fault Management

Richard C. Scalzo and Helmut Roth

Naval Surface Warfare Center Dahlgren Division
10901 New Hampshire Avenue
Silver Spring, Maryland 20903-5640

Abstract

This paper describes the top level of a meta-model for fault management. This meta-model can be used to help refine existing terminology. It can also be used to formalize standard development strategies for fault management services. This work is the outgrowth of a modeling effort that was performed by the Services for Reliable, Available, and Serviceable Systems Working Group (SRASS WG) which has been meeting as part of the IEEE Portable Operating System Interface (POSIX) Standards effort.

1. Introduction

The IEEE POSIX effort is aimed at producing a set of IEEE information technology standards. These are standards for application programs and user interfaces for open systems. Standards for application program interfaces (APIs) which address fault tolerance and serviceability issues are being developed by the SRASS WG. This WG has surveyed existing practice, obtained and examined base and reference documents, and made a preliminary list of APIs which are in common use. In addition, a process model for fault management was developed. The group has mapped APIs taken from base documents onto the process model. This was done to help refine group understanding of the semantics which underlie the fault management process.

The effort began with discussions to reach agreement on the terminology to be used. The terminology finally adopted as a baseline was that developed by the International Federation for Information Processing (IFIP) Working Group (WG) 10.4, [10]. In this paper, adherence to the terminology and the underlying semantics of IFIP WG 10.4 will be observed, wherever possible, and where it is not possible, the differences will be explained. In addition, interpretation of the terminology will be extended, wherever possible to include systems which contain components other than computers.

Existing terminology must often be refined in order to have the needed descriptive power. This has proven to be the case in the IEEE POSIX standards community. There is a need for systematic methods to address the terminology problem, especially in the area of fault tolerance. One important benefit to agreeing on and refining terminology is that a context is built for discussing the functionality of underlying services to be addressed. One way to improve terminology and to classify API functionality is to build a model that will make clear the services to be provided by the standard set of interfaces, the relations among them, as well as any interactions they may have. A brief description of the SRASS WG modeling effort and some extensions to it follows.

A meta-model is a model which describes a class of models. Executable implementations of meta-models may also be used to generate models in the class for which they are designed. If an object-oriented approach is used to implement a meta-model, each instantiation of the meta-model yields an object oriented model. The meta-model described here is an abstraction and translation of a process model developed by the SRASS WG. It is expected to provide a framework for further clarification of the terminology and help to identify the underlying semantics associated with APIs for fault management. The clarification of terminology adopted by the SRASS working group is now recognized as a necessary part of the standards development process.
1. SRASS Activities

The SRASS WG has defined fault management as the process of analyzing anomalous responses to requests for service (including both value and temporal errors) and of modifying the configuration of a system in order to improve system reliability and availability. The group identified two aspects of fault management: (1) services that support system serviceability (system maintenance and repair) and (2) services which support some aspects of fault tolerance, such as reconfiguration. Since an important part of the process of fault management is to provide fault tolerance, a brief discussion of the classification of methods used to make a system fault tolerant follows. A top-level classification of these methods is presented in Figure 1.

![Figure 1. Classification of Fault Tolerance Activities](image)

This classification was developed by the IFIP WG 10.4. Figure 1 shows that there are two types of methods needed to provide fault tolerance, (1) error processing and (2) fault treatment. The IFIP WG 10.4 defines error processing as the actions taken to remove errors from the computational state. The IFIP 10.4 WG defines fault treatment as the actions taken to prevent a fault from being reactivated, where an active fault is one which is producing an error.

The SRASS WG identified the top level functionality similar to that of the IFIP WG 10.4. These top level functions were identified as detection, diagnosis, and corrective action, as shown at the top of Figure 2. The term corrective action as used by the SRASS WG encompasses actions which support the recovery, passivation and reconfiguration activities in Figure 1.

The functional decomposition of fault management was carried forward to one level lower. These functions and the flow of information through them is depicted in Figure 2. These functions were identified as the minimum required to provide the top level fault management service: detection, diagnosis, and corrective action.

![Figure 2. Fault Management Process Model](image)

Note that the model uses the term event instead of error. The reason for this is that in order to provide reliable and available services it is sometimes important to detect anomalous responses to various components requests for service, as opposed to responses to requests for service which are erroneous. In fact, there is also a need to detect events which are a normal part of the operation of the system. For example, suppose that a hard disk experiences some abnormally long access times for a single read operation. If these long access times are not too frequent, then the disk remains operating within specifications and has not experienced a failure to return the data requested by the read operation. However, when the frequency of abnormally long response times reaches a certain threshold, this condition may indicate failure in the near future. Such abnormal responses need to be processed in order to prevent failures and the resulting loss of service or degradation of service. An example of an event which would not be classified as anomalous which would require detection is that of reconfiguration. The IFIP 10.4 WG uses the term "corrective maintenance" for activities aimed at removing faults.
before they are activated, thus preserving or improving a system's ability to deliver a service which complies with its specification. Repairing a disk drive once its response times become too long, but still within the system specification is an example of this.

The SRASS WG also made a preliminary mapping of some APIs to the process model. A list of some of the APIs, together with the top level functions that they map to, is shown in the list below. The API name appears on the left along with a short definition of the API's function together with an arrow showing which level 1 function from Figure 2 to which it was mapped. The mappings are grouped according to the operating system for which they were designed.

**SVR4 (Unix)**

- p_online: turn processor online/offline --- > reconfiguration
- processor_info: get info on one processor --- > reconfiguration
- sysconf: retrieve configurable system variables --- > reconfiguration
- pfmt, vpfmt: display error message in standard format --- > logging & notification
- addsev: define additional severities --- > logging & notification
- lfmt, vfmt: display error message in standard format and pass to logging and monitoring services --- > logging & notification

**Mach**

- task_ras_control:
  set pc's range of task's restartable atomic sequences --- > recovery
- processor_info: get info about processor --- > reconfiguration
- processor_control, processor_start, processor_exit: do something to a processor, exit a processor, start a processor --- > recovery, reconfiguration

---

**2. Description of the Meta-model**

The following description of the meta-model developed from the SRASS process model will consist primarily of descriptions of the object classes which the meta-model is meant to generate. Figure 3 depicts the top level of the fault management meta-model which was developed from the process model. Each box in the figure represents a class of objects and each arrow represents a class of relations. The class of environmental objects contains objects which are outside of the system. All of the objects in the Managed Object Class, and some of those in the Fault Management Class are contained within the system, while others such as humans may not be. Note that humans may be considered to be part of the system depending on their role in system operation. (If a human is acting as an operator of the system, then that human is considered to part of the system.)

---

Figure 3. Top Level Meta-Model
In addition, each instantiated object class should be considered to have some of the properties of objects in the Real-Time Object Model, RT0.k. [8] and [9]. An object in the RT0.k object model is an extension of the usual concept of objects. The most important extensions are as follows: (1) Each object has a real-time clock mechanism, and two types of methods: spontaneous (time triggered) and message triggered; (2) For each execution of a method an execution deadline associated with it; (3) Each real-time datum in an object also has a maximum duration for validity associated with it; (4) In addition there are consistency constraints two of which follow: (a) if replicas of a virtual real-time object are accessed at the same instant, then the time values delivered should never be off by more than one 'tick' of the virtual clock, and (b) whenever the position values of real-time object replicas in a virtual real-time object are visible to their clients, they must be the same. For the purposes of this paper, properties (1) - (3) will be adopted, but the set of consistency constraints will not be required.

There are three object classes which the model is meant to generate. An environmental object class, a managed object, object class and a fault management object class.

The environmental object class is meant to model the operational environment of the system. Since users are considered to be outside of the system there must be a relation to indicate service requests. In addition, the environment of the system generates events to which the system reacts which may not be requests for service. The environment may emit External Events to Managed Object Class, events such as room temperatures, which are out of specification for a given duration. Each of the subclasses of the environmental object class has time triggered methods when instantiated as well as event triggered methods. A Request for service from Managed Object Class may be event triggered, such as when a human interacts with the system, or they may be time-of triggered, such as when a signal is generated by a part of the environment, as in the case of a submarine in the ocean.

There are two subclasses which are not shown in Figure 3, the System User subclass and the External Event subclass. The System User subclass describes the rules of interaction of an end user with the system, so that a specification of the behavior of the user under various scenarios is included. The External Event subclass describes the rest of the environment, including the physical environment for the system together with any interactions or interference that may arise in the course of system operation.

The Managed Object, object class is an adaptation of an existing class hierarchy. The basis for the class hierarchies used may be found in [1]. This model is a modification of the class hierarchies in [1]. The Managed Object Class has four class hierarchies embedded within it: the Hardware Component Class, the Software Component Class, the Human Component Class, and the Complex Component Class. (Complex Components are components consisting of Hardware Components, Software Components, and Human Components together with relations describing certain interactions among them.) Top level diagrams for the class hierarchies for hardware, software, and human components are depicted in Figures 4-6 respectively. Each of these figures also shows examples of the simplest high level relations that are captured in the meta-model.

For reasons of readability, the Hardware Class Hierarchy has none of the relations defining the hardware architecture showing (Figure 4). In the examples below each of the relations is preceded by a source and followed by a destination. Some associative relations among the parts of the hierarchy include Processor is part of Complex Component, Link connects Processor to I / O Device, and Source Code is stored on Storage Device.

![Figure 4. Top Level Hardware Hierarchy](image-url)
are expressed. Other relations subclasses and relations are not displayed. There are also several associative relations that the meta-model uses. Among these are the relations Source Code is being modified by Software Engineer, Executable Code is being repaired by Software Engineer, Source Code is stored on Storage Device, and Executable Code is stored on Storage Device, as well as Executable Code is allocated to Processor.

Figure 5. Top Level Software Hierarchy

The class hierarchy for the human components of a system is shown in Figure 6, where no relations are depicted. However, examples of relations that are modeled are Operator reconfigures Managed Object and Hardware Engineer repairs I/O Device.

Figure 6. Top Level Human Hierarchy

The Fault Management Class describes the manner in which the components interact when there are anomalous events that need to be accounted for, especially when the anomalous events affect more than a single component. In particular, system observability characteristics, error containment regions, and the mode of system operation are described here. Since logging and notification are handled by internally by Hardware Components, Software Components, and Human Components, the anomalous event logging and notification strategy are described here.

3. Example of a Simple Use of the Meta-Model

In this section a brief description of the use of the meta-model is made. The meta-model has provided a framework for the refinement of the SRASS terminology and, in turn refinement of the terminology has led to refinement of the model. Two very simple examples of the interplay between terminology and meta-model are given. This interplay is continuing as the meta-model and the terminology are refined. The two examples given below were the result of a more detailed examination of the preliminary mapping of APIs to the SRASS process model in concert with a preliminary version of the meta-model.

In the first example, three APIs from Unix SVR4 were examined: (1) p_online, (2) processor_bind, (3) processor_info, and (4) sysconf. The class hierarchies for the Managed Object Class suggested that reconfiguration could be decomposed into hardware reconfiguration, software reconfiguration, human component reconfiguration, and complex component (system) reconfiguration. For the purposes of POSIX, the terminology was refined only for software, hardware and complex components. These hierarchies, together with the RTO.k object model also suggested that the processor bind operation should be mapped onto complex component reconfiguration, because some processes may fail to meet maximum execution time constraints on some processors, especially when a system processor heterogenous. Thus processor bind has the potential to provide some measure of fault tolerance to the system. In addition, it is clear from the hierarchies that the allocation of executable software to a processor, may be considered to be a kind of reconfiguration of the system. The allocation of executable software to a processor, must also be accompanied by allocation to permanent storage and memory, consequently there are several types of software reconfiguration to be considered.

Another refinement of the SRASS terminology arose as a result of a more detailed examination of the Mach API task_ras_control. In this case this examination also
led to a refinement of the meta-model, when it was noted that this API provides an interface to the Mach kernel and that it requires the concept of software execution scheduling. Some software dependency relations had already been identified in the preliminary version of the meta-model, these, however, were all static or very high level, such as executable software is stored on disk, or executable software is running on hardware module. Thus other software relations were added to the meta-model, e.g., the Runs before and Is scheduled before relations. In addition, the top level recovery function was refined to take into account various levels of recovery, e.g., recovery at the task level and recovery at the thread level.

Other potential uses for the model are being investigated. Among these are a means of classifying the top level characteristics of fault management strategies for various system vendors. The aim is to provide a framework for classifying various approaches to providing fault management services for purposes of identifying comparison.

In the following lists a slightly more detailed description of the object classes associated with the meta-model are given. For the purposes of conciseness, descriptions of the classes below use a template recommended in [2].

**Class Name: Environmental Objects**

**Subclass Names:**

(1) System User
(2) External Event

**Attribute Classes:**

(1) System User: Name of system user, System user characteristics: Types of service requests, Permissions, Connection status, Max wait time for response
(2) External Event: Name of external event generated, Time of external event generation, External event characteristics: System components affected

**External Inputs:**

(1) System User: Type of user service received, Time service received
(2) External Event: None

**External Outputs:**

(1) System User: Type of service requested, Time of request
(2) External Event: Name of external event generated, Time of external event generation, External event characteristics

**Service classes:**

(1) System_User: Send request for service, Receive service
(2) External Event Generator: Generate external event, Send anomalous external event to affected components

**Class Name: Managed_Objects**

**Subclass Names:**

(1) Hardware Component
(2) Software Component
(3) Human Component
(4) Complex Component

**Attribute Classes:**

(1) Hardware Component: Name of hardware component, Characteristics of hardware component, State of hardware component, Max time allowed in this state
Note: Subclasses (2), (3), and (4) have the same high level attributes as Hardware Component

**External Inputs:**

(1) Hardware Component: Type of external event received, Time of external event, External event characteristics, Component user service request
Note: Subclasses (2), (3), and (4) have the same high level external inputs as Hardware Component

**External Outputs:**

(1) Hardware Component: Type of response to service request, Type of response to event
Note: Subclasses (2), (3), and (4) have the same high level external outputs as Hardware Component
Service Classes:

(1) Hardware Component: Type of event generator, Type of passive event detection, Type of error recovery, Type of error signalling
(2) Software Component: Same as Hardware Component, plus Type of active event detection, Type of symptom encoding, Type of symptom collection, Type of event logging, Type of event notification, Type of event classification, Type of recovery event generation, Type of reconfiguration event generation
(3) Human Component: Type of passive event detection, Type of recovery event generation, Type of reconfiguration event generation, Type of event classification

Class Name: Fault Management Objects

Subclass Names:
None

Attribute Classes:

System level fault tolerance characteristics, System level serviceability characteristics, System level fault tolerance strategy, System level serviceability strategy, System state, Max time allowed in this system state

External Inputs:

Hardware Component attributes, Software Component attributes, Human Component attributes, Type of event occurrence, Time of event occurrence, Event characteristics

External Outputs:

Type of system level detection events, Type of system level reconfiguration events, Type of system level recovery events

Service Classes:

Type of system level reconfiguration action, Type of system level recovery action, Type of status message to system users, Type of status message to system components

A complete specification is in preparation. In order that the specification be in a format suitable for compilation, the authors are using the meta-modeling rules adopted by the Electronic Industry Association Case Data Interchange Format (EIA CDIF) effort, [4] - [6]. In addition, the specification is being prepared using the EIA CDIF specification format.

References: