

Atacama Large Millimeter

ALMA-SW-NNNN

Revision: 1.4

2007-11-26

ACS Alarm System

Software Architecture and How-to manual

Alessandro Caproni (acaproni@eso.org),

ESO

Keywords:

Author Signature:Date:

Approved by:Signature:Institute:Date:

Released by:Signature:Institute:Date:

Table of Contents

1 Introduction	4
1.1 Glossary.....	4
1.2 References.....	5
2 Installation and test	6
3 AS architecture	7
3.1 The resource tier.....	8
3.2 The business tier.....	12
3.3 The client tier.....	13
4 CDB configuration	14
4.1 AlarmDefinitions.....	16
4.2 Administrative.....	18
4.3 AlarmSystemConfiguration.....	19
4.4 Categories.....	20
4.5 ReductionDefinitions.....	22
5 The alarm GUI	25
6 The CERN operator GUI	32
7 ACS implementation	37
7.1 Alarm source client.....	38
7.2 Alarm category client.....	38
7.3 alarmSourcePanel.....	39
7.4 BACI properties.....	40

1 Introduction

The alarm system (AS) is a messaging system: it collects, manages and distributes information about abnormal situations and shows what's happening to the user.

In a complex environment, a malfunctioning, either in the software or the hardware, might trigger a chain of malfunctions in other equipment. The main purpose of the alarm system is to help the user to identify the root cause of a problem in such a way that he can fix the problem in a short time. This is achieved comparing the alarms active at a given time against a knowledge base describing the correlation between the alarms.

The alarms are sent by the sources to the Alarm Service Component (ASC) whenever an abnormal situation is detected. The ASC listens for the alarms and when a new alarm is received it looks in its knowledge base to see whether a correlation exists between the new alarm and the other alarms already active. If such a correlation is found then the ASC could mask/hide some alarms that are not relevant for identifying the root cause of the malfunction. Finally, the ASC builds a new version of the alarms to send to the clients. This new version is more complete and human readable than the alarms generated by the sources.

The AS has a set of clients; one of them is the operator GUI that shows the alarms to the operator.

In the distribution there are two alarm systems:

1. the LASER Alarm System, the subject of this document.
2. an ACS implementation of the Alarm System that logs a message for each alarm published

The ACS implementation is used as default: you can see the alarms with `jlog` or the `loggingClient`. To switch to the CERN implementation, the `Alarms` branch must be present in the CDB and the `Implementation` property must be explicitly set to `CERN`¹.

The development of the AS is an ongoing process. The interfaces for the developers should remain untouched. However the AS is not complete and some part of the system may not be fully functional or not implemented yet. In particular, the ACS LASER AS derives from the AS developed at CERN for the Large Hadron Collider (LHC). At the present, the porting of the original AS is not complete and some parts of the original AS are missing.

1.1 Glossary

ACS Alma Common Software

¹You can find more about the CDB for the alarm system in paragraph 4.

ALMA Atacama Large Millimeter Array

AS Alarm System

ASC Alarm Service Component

CDB Configuration DataBase

CERN European Organization for Nuclear Research

FC Fault Code

FF Fault Family

FM Fault Member

FS Fault State

GUI Graphical User Interface

JMS Java Message Service

LASER LHC Alarm SERVICE

LHC Large Hadron Collider

MR Multiplicity Reduction

NC Notification Channel

NR Node Reduction

RR Reduction Rule

See also:

<http://www.alma.nrao.edu/development/computing/docs/joint/draft/Glossary.htm>

1.2 References

- ACS documentation:
<http://www.eso.org/projects/alma/develop/acs/OnlineDocs/index.html>
- LASER project: <http://proj-laser.web.cern.ch/proj%2Dlaser/>
- LHC control project: <http://lhc-cp.web.cern.ch/lhc%2Dcp/>

2 Installation and test

The AS is part of the ACS distribution. It is composed of several modules: the modules of the ACS implementation of the AS that are built and installed before the containers and the CERN implementation of the AS whose modules are grouped into ACSLaser and compiled just after ACS.

The AS is in CVS project2: `ACS/LGPL/CommonSoftware/ACSLaser`.

There is a little demo that can be used to test the functioning of the AS. The demo shows the functioning of the Node Reduction and the Multiplicity Reduction. A complete CDB is available for this purpose in the `demo` module.

The NR demo consists of three modules capable of sending alarms: `ALARM_SOURCE_PS`, `ALARM_SOURCE_MOUNT` and `ALARM_SOURCE_ANTENNA`. When the power supply fails, it sends an alarm and triggers a failure of the mount component. In the same way a failure in the mount components causes the sending of an alarm and the failure of the antenna component. The purpose of this demo is to show how the alarms are reduced and only the root cause of the problem (the alarm in PS) is shown in the GUI.

The MR demo consists of a single component, `MULTIPLE_FAILURES`, sending 5 different alarms when the `multiFault` IDL method is executed. The threshold for the MR is set to 3. The purpose of this demo is to show how several alarms of the same kind are reduced when the number of such active alarms is greater than the threshold.

Follow these steps to check your installation and execute the NR demo:

1. set `ACS_CDB` to the CDB folder in `ACSLaser/demo/test`
2. start ACS: it will start the ASC
3. start the java container, `frodoContainer`
4. start the java container `javaContainer`²
5. start `objexp`
6. start the `ALARM_SOURCE_PS` component
7. start the `ALARM_SOURCE_MOUNT` component (be careful because there are 3 mount components, one for java, one for C++, called `ALARM_SOURCE_MOUNTCPP` and one for python, called `ALARM_SOURCE_MOUNTPY`)
8. start the `ALARM_SOURCE_ANTENNA`
9. start the GUI with the command `alarmPanel`; the panel `panel` has a green icon in the right bottom side of the main window meaning that the application is connected to the AS

² There are 2 java containers. The power supply and the mount run in `frodoContainer`; the antenna and `MULTIPLE_FAILURES` components run in `javaContainer`.

10. from the `objexp`, select the PS component and execute its fault function.

The execution of the PS `fault_PS` IDL method, causes the three components to send a chain of alarms that appear in the main window of the GUI³. If the reductions are turned on, you will see only one the alarm generated by the power supply failure i.e. the root cause of the chain of failures. If the reductions are disabled, all the alarms will appear in the GUI. A button in the toolbar of the panel allows to enable/disable the alarm reductions in the table.

To execute the MR demo instead of the NR, in step 8 activate `MULTIPLE_FAILURES` and execute its fault method.

In both examples, the terminate fault method will set all the alarms from `ACTIVE` to `TERMINATE`. The color of the alarms in the GUI will change to green indicating that the abnormal situation has been fixed.

The component `ALARM_SOURCE_MOUNTCPP` is a C++ component running inside the C++ container `bilboContainer`. It is there to show how to send alarms using C++ API but it is not configured for reduction.

The component `ALARM_SOURCE_MOUNTPY` is the demo of an alarm generated by a python component.

The ASC submits a great number of log messages when reading the CDB. These messages are easy to identify filtering for the source object `AlarmService`.

When the sources send alarms a message is also logged at alert level.

3 AS architecture

The AS is a distributed, layered application. Each layer depends on the layer below and provides a set of services for the layer above. The definition of a clear set of interfaces drives this hierarchy.

As shown in fig.1, The software is composed of three tiers: the *resource tier*, containing a set of sources, to detect malfunctions and send alarms; the *business tier*, that implements the system logic and its services, having a knowledge base of the specific domain; the *client tier* composed of clients consuming the business services.

³ To simulate a chain of failure, each component sends an alarm, waits 5 seconds and calls the fault method of the next component. This mean that the fault method of PS needs about 15 secs to terminate. The sleep time of 5 secs between the sending of an alarm and the execution of the IDL method of the next component of the chain has been introduced to help the user to look at the GUI and see the changes when each new alarms arrive. Objexp has a timeout of 5sec for each IDL method executed on a component so it will show a dialog saying that the fault method of PS has caused a timeout. To avoid this notification you should extend the timeout setting properly the `objexp.pool_timeout` property as described in the Object Explorer user manual.

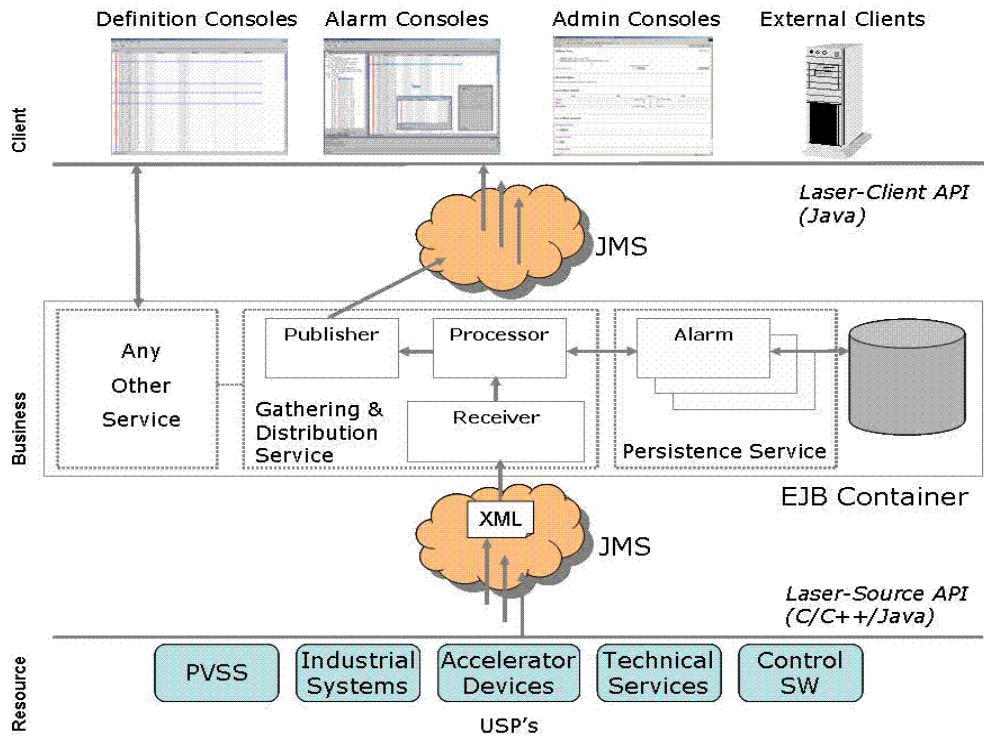


Figure 1: The architecture of the LASER alarm system

3.1 The resource tier

The resource tier is composed of the sources of alarms, i.e. applications that monitor the hardware and the software to detect malfunctioning. The sources can be written using different programming languages and run on different platforms.

Each alarm is identified by a Fault State (FS). A FS is a triplet: the Fault Family (FF), the Fault Member (FM) and the Fault Code (FC).

$$FS = \langle FF, FM, FC \rangle$$

The FF is a string and identifies a set of elements of the same kind, for example the set of all the power supplies. The Fault Member, again a string, identifies a particular instance of the elements of the FF, for example PS3 i.e. a power supply, called PS3. The FC is an integer describing the specific fault, for example 100 for over current.

Conversely speaking, a FS is a unique identifier for an alarm. It means that for each possible alarm that can be generated by each possible source there is one and only one FS. And vice versa, to each alarm corresponds one and only one triplet. The triplet does not say *where* an alarm happens but it says *which* alarm is occurring. The information *where* must be inserted in the database and we'll talk about that later.

These are examples of valid triplets:

- <MOUNT, MOUNT_1, 100> (ACS component)
- <ANTENNA, ANTENNA7, 200> (Hardware equipment)
- <kernelModule, pcnet32@host, 1> (kernel module)

In the examples, MOUNT_1 is the name of a component. Each component can easily retrieve its name by calling the `getName` method of the `ContainerServices`. `pcnet32` is the name of a kernel module, and `host` is the name of the host where the module is running.

The triplet identifies an alarm sent by a source to the ASC. The triplet is also used internally by the ASC to apply the reduction rules or retrieve more information about a specific alarm from the database. We can think at each triplet as a unique identifier used in low level computation. The information sent by the ACS to the clients, the operator GUI for example, is not a triplet as it is received by a source but is a more human readable and complete description of the alarm represented by that triplet.

There are no guidelines yet for the strings and numbers to use to define the triplets but common sense suggests to use meaningful strings for FFs and FMs and possibly coherent numbers for failures of the same kind. For example, for a component the FF could be its IDL interface and the FM could be its name. For a kernel module, the FM could contain the name of the module and the name of the host where the module is executed.

An FS, i.e. an alarm, has a status, active or inactive. When an abnormal situation occurs, a source sends an active alarm. When the problem has been fixed, the source sends the same alarm with status inactive to the ASC. When a source is created, there is no need to send an alarm, even an inactive one; but when an abnormal situation has been fixed and the alarm does not exist anymore, the source must send an inactive alarm.

An heartbeat mechanism is associated with a source so that in case of a problem so bad that the source itself crashed before sending an alarm, the ASC is able to detect that something happened to the source because of the missing receipt of the heartbeat.

The Laser-source API has been written to connect the sources to the business tier and is very small in order to be as simple as possible for the user. The API is written in java, C++ and python. The following shows a java example to send an alarm.

```
import alma.alarmsystem.source.ACSAlarmSystemInterfaceFactory;
import alma.alarmsystem.source.ACSAlarmSystemInterface;
import alma.alarmsystem.source.ACSFaultState;
...
public void send_alarm(
    String faultFamily,
    String faultMember,
    int faultCode,
    boolean active)
{
    ACSAlarmSystemInterface alarmSource =
        ACSAlarmSystemInterfaceFactory.createSource(this.name());
    ACSFaultState fs =
```

```

    ACSAlarmSystemInterfaceFactory.createFaultState(
        faultFamily, faultMember, faultCode);
if (active) {
    fs.setDescriptor(ACSFaultState.ACTIVE);
} else {
    fs.setDescriptor(ACSFaultState.TERMINATE);
}
fs.setUserTimestamp(new Timestamp(System.currentTimeMillis()));
Properties props = new Properties();
...
props.setProperty(...);
fs.setUserProperties(props);
alarmSource.push(fs);
}

```

The following shows a C++ example to send an alarm:

```

#include "ACSAlarmSystemInterfaceFactory.h"
#include "AlarmSystemInterface.h"
#include "FaultState.h"
#include "faultStateConstants.h"
#include "Timestamp.h"
#include "Properties.h"

using namespace acsalarm;
...

// constants we will use when creating the fault
string family = "AlarmSource"; // FF
string member = getComponent()->getName(); // FM, the name of the component
int code = 1; // FC

// Create the AlarmSystemInterface using the factory
auto_ptr<AlarmSystemInterface> alarmSource =
    ACSAlarmSystemInterfaceFactory::createSource();

// Create the FaultState using the factory
auto_ptr<acsalarm::ACSFaultState> fltstate =
    AlarmSystemInterfaceFactory::createFaultState(family, member, code);

// Set the fault state's descriptor
string stateString = faultState::ACTIVE_STRING;
fltstate->setDescriptor(stateString);

// Create a Timestamp and use it to configure the FaultState
Timestamp * tstampPtr = new Timestamp();
auto_ptr<Timestamp> tstampAutoPtr(tstampPtr);
fltstate->setUserTimestamp(tstampAutoPtr);

// Create a Properties object and configure it, then assign to the FaultState
Properties * propsPtr = new Properties();
propsPtr->setProperty(faultState::ASI_PREFIX_PROPERTY_STRING, "prefix");
propsPtr->setProperty(faultState::ASI_SUFFIX_PROPERTY_STRING, "suffix");
propsPtr->setProperty("TEST_PROPERTY", "TEST_VALUE");
auto_ptr<Properties> propsAutoPtr(propsPtr);
fltstate->setUserProperties(propsAutoPtr);

// Push the FaultState to the alarm server
alarmSource->push(*fltstate);

```

The following is a python example to send an alarm.

```
import Acsalarmpy
import Acsalarmpy.FaultState as FaultState
import Acsalarmpy.Timestamp as Timestamp

Acsalarmpy.AlarmSystemInterfaceFactory.init()

alarmSource=Acsalarmpy.AlarmSystemInterfaceFactory.createSource("ALARM_SYSTEM_SOURCES")
fltstate=Acsalarmpy.AlarmSystemInterfaceFactory.createFaultState(family,member,code)

fltstate.descriptor = FaultState.ACTIVE_STRING

fltstate.userTimestamp = Timestamp.Timestamp()
fltstate.userProperties[FaultState.ASI_PREFIX_PROPERTY_STRING] = "prefix"
fltstate.userProperties[FaultState.ASI_SUFFIX_PROPERTY_STRING] = "suffix"
fltstate.userProperties["TEST_PROPERTY"] = "TEST_VALUE"

alarmSource.push(fltstate)

Acsalarmpy.AlarmSystemInterfaceFactory.done()
```

The sources build a message containing the FS and an action, like active or terminate. The API embeds the message in a structure and publishes the message in a JMS topic to the business tier.

To define an alarm as active or inactive, the `setDescription` method must be used passing the right string. It is important to remember that the descriptor of an alarm is the status of the alarm itself and not a generic string to describe what's happening at a certain instant. In fact, the description of the alarm is stored in the database.

In java, the descriptor of the alarm must be one of the constant String of `alma.alarmsystem.source.ACSFaultState` (for example `ACSFaultState.ACTIVE`). In C++, the descriptors are defined in `faultStateConstants.h` under the `faultState` namespace (like for example `faultState::ACTIVE_STRING`).

It is a common mistake to set the descriptor of an alarm to a user defined string in the `setDescription` method. In that case the alarm system will ignore the alarm (i.e. the alarm will be discarded as unrecognized and it will not be processed and published).

The alarms produced by the sources are sent to the ASC through JMS. It is possible to configure the AS in order to use more channels and it is also possible to define which channel is used by each source to send an alarm to the ASC⁴.

⁴In the present version of the AS, all the alarms are published in the same channel named `ALARM_SYSTEM_SOURCES`.

The ASC must receive all the existing alarms so the ASC listens for alarms from all the source channels.

There are two implementations of the sources. One uses the CERN implementation i.e. the sources send alarms to the ASC by means of JMS. Alternatively, it is possible to use the ACS implementation i.e. the sources log a message for each alarm sent. The `ACSAlarmSystemFactory` generates objects for the requested implementation in a transparent way. The implementation to use can be chosen by changing a property in the CDB⁵.

3.2 The business tier

The business tier is the core of the alarm service:

- listens for FS changes and heartbeats from the sources
- reads the further data of a received alarm from the database
- reduces or masks the FS depending on the knowledge of the environment and the current status of the system
- persists the FS
- traces and archives the changes of the FS
- allows management of the alarm system without stopping the alarm service component
- authenticate users on the client GUIs

All these services⁶ are realized by the ASC and the communications between the upper and the bottom layers happen through a definite API.

The persistence, trace and archiving of FS are not implemented in this version of the AS.

In order to keep the Laser-source API simple, a source sends to the business layer only the triplet describing the alarm with the time of its creation and its state. For each alarm received, the AS reads its complete definition from the CDB in order to present a complete snapshot of the situation, its possible solution and consequences to the operators. The following table shows some of the information stored in the database for each alarm⁷.

system-name	The name of the system	identifier	The identifier of the system	problem-
description	The description of the problem	cause	The cause of the problem	action
	The action the operator must follow to fix the problem	consequence	A description of the consequences of the alarm	priority
	The priority of the alarm: there are four priority levels	contact name	The name of the responsible person	contact gsm
	A GSM number to call			

⁵See chapter 6.

⁶Not all the services have been ported from the CERN version of the AS to the prototype. In particular the prototype doesn't implement the persistence of the FS, the FS changes and the users configuration storage. This is due to the lack of the database.

⁷The triplet is the unique identifier to get the informations shown in the table from the database. The data in the table should be more interesting then the triplet for the operators.

to notify about the problemcontact emailThe email address of the responsible person for the problemhelp-urlAn url that point to the documentation of the problemsource-nameThe name of the sourcelocationThe location of the source

One of the most relevant parts of the business tier is the reduction of the alarms. In a complex environment where a failure can cause a cascade of secondary alarms, it is very important to show to the operators the root cause of a problem. Operators are also confused when the operator GUI shows a great number of repeated alarms of the same type. The alarm reduction mechanism addresses both these problems.

To perform the reduction, the alarm system reads from the database a set of dependency rules between alarms describing their correlation. Whenever the service receives a FS change, it applies that set of rules and eventually marks some alarms as reduced.

All the alarms, both reduced and not reduced, will be sent to the client because some clients may be interested in receiving all the alarms regardless of their reduction status: it is the GUI that hides the reduced alarms to the operators depending on the specific configuration.

There are two types of reduction rules:

- *node reduction*: when it is known that a failure in an equipment A triggers a failure also in the equipment B then the latter alarm is reduced, with the effect that only A, the root cause of the FS, is shown;
- *multiplicity reduction* when there is a great number of alarms of the same type⁸ then these alarms are reduced and a new alarm is shown effectively reducing the number of alarms shown in the client GUI.

Four priority levels (0 to 3) are foreseen for the gravity of the alarms. Different priority levels are shown with different colors in the operator GUI.

3.3 The client tier

The client tier consists of java applications that consume the data published by the business tier. The client connects to the business tier by means of the Laser-console API . The business tier supports both login and configuration facilities⁹.

Once connected, the clients can access services of the business tier by means of the laser-client API .The communication between the alarm service and the client applications happens through JMS whose implementation is based on Notification Channels (NC).

The alarms may be subdivided in categories and to each category corresponds one and only one NC. When the alarm service receives an alarm, it retrieves the name of its

⁸ The MR is not activated by sending several times the same ACTIVE alarm. The MR is activated when a set of *different* alarms are all active at a certain instant of time and their number is greater then a given threshold. The alarms that must be counted as part of a MR is defined in the CDB.

⁹In the prototype the only existing user is test. The configurations are not stored in the database so the user has to setup his configuration every time he logins.

category together with other information for the operator from the database. The alarm service checks the alarm against the actual situation and the reduction rules and finally sends the alarm to the application of the client tier publishing the alarm in its category/NC.

Three GUIs developed with Netbeans are part of the client tier: the definition console, the alarm console and the admin console¹⁰. The definition console and the admin console allow the user to define alarms, sources and categories as well as create accounts and configurations for the operators of the alarm console. In this version of the AS the only available GUI is the alarm console, described in chapter 6.

The Alarm console shows the alarms to the operators: when the operator starts the GUI, he has to log into the system, the GUI then loads his specific configuration and connects to the alarm service. In the configuration it is possible to select the categories of the alarms to show to the operator, showing him only the alarms relevant for his area of interest. In the configuration it is also possible to define whether the operator is interested in receiving all the alarms or only the reduced alarms. An apposite configuration panel allows the user to change its configuration at run-time.

alarmPanel is an operator GUI developed in java for ACS. It shows alarms from all the categories in a small panel that can be run in stand alone as well as inside OMC. The application connects to the alarm service to know all the existing categories and connects to the notification channels. A view of the active and inactive alarms is given by means of a a SWING table.

4 CDB configuration

Each alarm must be defined in the CDB in order to be recognized and managed by the ASC. It is possible to define default alarms and categories to shorten the effort of setting up the CDB. However you should keep in mind that the main purpose of the alarm system is to present to the operators detailed information about an alarm in order he/she can fix the problem very soon. Using default definitions allows to have a shorter CDB but generic informations are probably not enough to present the operators all they need to fix a problem. Therefore we suggest to use default values as less as possible and paying attention that the informations presented by default alarms fit with the real situation.

The definition of each alarm is very important because it contains both the information for the alarm system engine and those for the client tier applications that should be complete and clear enough to show to the operators all the information they would need to fix an abnormal situation.

A misconfiguration of the CDB does not inhibit a source from publishing an alarm as can be easily verified running the alarmSourcePanel or looking at the logs. A

¹⁰Only the operator GUI, the alarm console, have been ported in the prototype.

misconfiguration instead does not allow the ASC to send alarms to operators because there is no way for the service to know on which category an undefined alarm must be published.

The definition of the alarms is entirely contained in the Alarms folder in the CDB having the following structure:

```
CDB
...
Alarms
  Administrative
    AlarmSystemConfiguration
    Categories
    ReductionDefinitions
  AlarmDefinitions
    FaultFamily_1
    FaultFamily_2
    ...
    FaultFamily_n
```

The schemes defining the content of XML files are in the `acsalarmidl` module. In `ws/config/CDB/schemas` you can find the following files:

- `acsalarm-categories.xsd`: is the schema for the XML file into `Categories` folder
- `acsalarm-fault-family.xsd`: contains the definition for XML files called `FaultFamily_x` in the example upon
- `AcsAlarmSystem.xsd`: is used by `AlarmSystemConfiguration` and `ReductionDefinitions`

We describe the content of the `CDB/Alarms` folder in the same order the developers should follow to populate the database.

The `Alarms` folder must exist in order to use the CERN alarm system. `Alarm/Administrative/AlarmSystemConfiguration/AlarmSystemConfiguration.xml` must also exist and have the “implementation” property set to CERN. If one of the previous conditions is missing, ACS will not use the CERN AS. Note that ACS log a message with the alarm system in use to help debugging.

During activation, the ASC logs a large number of debug messages for every piece of information it reads from the CDB: `jlog` can be of a great help while setting up the CDB for the alarms¹¹.

Alarms is composed of two folders:

¹¹ To use `jlog` for this purpose you should remember to set the log level to `trace` and discard level to `none`. Filtering by source name (`AlarmService`) is also suggested.

- `Administrative` that contains all the administrative informations like the definition of the reduction rules and the categories to group the alarms;
- `AlarmDefinitions` with the definition of all the alarms.

Each time a new alarm is created, its definition must be added in `AlarmDefinitions`. The alarm system administrator deals with the `Administrative` part of the CDB defining how to group alarms into categories and how reduce/mask alarms. The administrator defines how and to whom the alarms will be shown.

4.1 AlarmDefinitions

The first step while creating alarms is the definition of the triplets. As we said before the FF is usually the IDL of the components while the FM is the name of a specific component implementing such interface. The FC represents the specific alarm sent by such a component. This definition includes dynamic components whose name (and therefore the FM) is known only at run time.

Referring to the demo example we have PS, Antenna, Mount and MF idl interfaces and a bounce of components implementing such interfaces.

While creating alarms for a component, the developer must create a folder with the name of the fault family (i.e. the IDL of the component), having an XML file with the same name inside:

```
CDB
  Alarms
    AlarmDefinitions
      Mount
        Mount.xml
```

It means that we have only one folder for each FF, i.e. one folder for each IDL definition. Inside such folder, there is only one XML file containing the definition of all the fault codes (i.e. the type of alarms sent by the component implementing such interface) and the definition of all the fault members (i.e. the component names, including dynamic components).

The definition of the alarms sent by components implementing the Mount IDL interface is described by `Mount.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<fault-family name="Mount" ...>
  <alarm-source>ALARM_SYSTEM_SOURCES</alarm-source>
  <help-url>http://tempuri.org</help-url>
  <contact name="Ale"/>
  <fault-code value="1">
    <priority>1</priority>
    <problem-description>Mount test</problem-description>
  </fault-code>
```

```

<fault-member-default>
</fault-member-default>
<fault-member name="ALARM_SOURCE_MOUNT">
</fault-member>
</fault-family>

```

The document type is `fault-family` having the attribute `name` with the name of the fault family defined by the file¹², `Mount` in our example.

The following table describes the tags of the XML.

XML tag name	Tag meaning
<code>alarm-source</code>	The name of the NC in use by the sources while sending alarms to the ACS for the Mount components. This field is fixed and therefore can't be changed by the developer since in current version of the alarm system only one channel is supported for all the sources ¹³ .
<code>help-url</code>	The URL where the operators find detailed information about the alarms published by the components of the family. The web page should contain at least a detailed description of all the alarms and the way to fix problem.
<code>contact</code>	Defines the person to contact in case the operator needs more information to fix the situation. When an alarm is raised for this family, the contact person should always be informed of the malfunction. This tag contains three other tags <ul style="list-style-type: none"> ● <code>name</code> ● <code>email</code> ● <code>gsm</code>
<code>fault-code</code>	The XML contains a non-empty array of fault codes. They are the FCs of the triplets of the family. All the possible FCs must have an entry of this type. The attribute <code>value</code> , is the number of this FC and is mandatory. <code>fault-code</code> contains the following tags: <ul style="list-style-type: none"> ● <code>priority</code> is an integer in the interval [0,3] ● <code>cause</code>: the cause of the problem ● <code>action</code>: the action to fix the problem ● <code>consequence</code>: possible consequence of the problem ● <code>problem-description</code>: a short description of the problem
<code>fault-member-default</code>	The definition of the fault member to use when a <code>fault-member</code> is not defined for a given triplet. <code>fault-member-default</code> contains an optional tag <code>location</code>
<code>fault-member</code>	The file contains an array of <code>fault-member</code> each of which represents a FM of the triplet.

¹² It is the FF of the triplet.

¹³ In future versions of the alarm system, it will be possible to have more than one channel for different sources. We therefore preferred to force the definition of the source channel in the XML.

	The tag has a mandatory attribute name that is the FM of the triplet (i.e. the name of the component sending the alarm). It is possible to define a <code>location</code> as element of this tag.
--	--

`fault-member-default` is the default member definition used by the ASC when it receives a triplet whose FM is not described by any `fault-member` entry of the family. When a triplet is received, the ASC reads the default definition to fill the fields of the alarm description, leaving the FM name as it is in the triplet. It is possible to define a `location` for a `fault-member-default` but it is optional because sometimes it is not possible to define a location (like for example for dynamic components).

The `fault-member-default` is normally used for dynamic components whose names (and therefore their FMs) are unknown before run-time. Another situation where the definition of a default member can be very useful is the definition of BACI properties as described in chapter 7.4.

The demo example shows the usage of `fault-member-default` for the component `ALARM_SOURCE_MOUNT_CPP`.

A `location` is composed of four string elements:

- `building`: the building of the failing equipment
- `floor`: the floor in the building
- `room`: the room number or name
- `mnemonic`: a mnemonic to quickly recall the position of the equipment
- `position`: the exact position in the room

Reassuring, the definition of alarms is done having the fault family in mind. The developers must define one folder for each fault family containing one XML file with the definitions of all the members and the codes of the alarms sent by the members of the family. The fault codes describe the type of the alarms and there must be at least one fault code. Fault members represent the entities that can produce alarms but in this case it is possible to define a default member avoiding the repetition of the same definition several times in the file.

4.2 Administrative

This part of the CDB defines how alarms are presented to the operators resembling the process of creating views by a database administrator.

As we said before, the developer describes the meaning of each alarm by filling the `AlarmDefinitions` part of the CDB. Whenever the ASC receives a triplet from a source, it reads the definition of the alarm from this part of the CDB.

After reading the alarm definition, the ASC starts processing the alarm checking if some of the reduction rules is applicable and eventually publishes the alarm in one or more categories where the clients listen to alarms. If an alarm is not associated to any of the

existing categories then it will not be published and therefore will not be visible by ASC clients like the operator GUIs..

Categories represent the mechanism used by the administrator to create different views of the alarms: he/she decides in which categories an alarm will be published. Users listens to alarms from the categories they are allowed to connect¹⁴.

4.3 AlarmSystemConfiguration

AlarmSystemConfiguration contains AlarmSystemConfiguration.xml:

```
AlarmSystemConfiguration
  AlarmSystemConfiguration.xml
```

This XML file contains a list of properties common to the whole AS. At the present there is only one property that can be defined: Implementation.

<i>Property name</i>	<i>Property values</i>
Implementation	<ul style="list-style-type: none">● CERN if the AS uses the CERN implementation● ACS: if the AS uses the ACS simple implementation i.e. the sources log a message instead of sending messages to the ASC <p>If this property is not found or it has a wrong value, then the ACS implementation of the alarm system is used.</p>

ACS comes with two implementations of the AS: ACS and CERN. The ACS implementation logs a message for each published alarm. The message basically consists of the triplet and the status of the alarm. It is possible to browse the alarms with jlog , or the loggingClient.

CERN implementation is the subject of this document. The following steps are needed to activate the CERN implementation of the AS:

1. define the Alarms branch of the CDB as described in this chapter
2. set the Implementation property of AlarmSystemConfiguration.xml to CERN

You can switch from one implementation to the other simply changing the Implementation property of the CDB i.e. you do not need to remove the Alarms branch if it is present.

The format of the XML file is the following:

¹⁴ The CERN operator GUI allows the user to select the categories they want to listen to alarms from. The alarmPanel instead automatically connects to all available categories.

```
<?xml version="1.0" encoding="UTF-8"?>
<alarm-system-configuration xmlns="urn:schemas-cosylab-com:Alarm:1.0"...
  <configuration-property          name="Implementation">CERN</configuration-
property>
</alarm-system-configuration>
```

Each property is defined with the tag `configuration_property` and the name of the property is in the attribute name.

4.4 Categories

The Categories folder contains `Categories.xml`:

Categories

`Categories.xml`

Here are defined all the categories (and therefore all the notification channels) into which the alarms are grouped for publishing by the ASC. The categories are used by the applications of the client tier. For example each user of the operator GUI has a configuration with the categories he/she is authorized/interested in.

This is the configuration of categories from the demo example:

```
<categories
  xmlns="urn:schemas-cosylab-com:acsalarm-categories:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <category is-default="true" path="CATEGORY1">
    <description>Test category 1</description>
    <alarms>
      <FaultFamily>Antenna</FaultFamily>
      <FaultFamily>Mount</FaultFamily>
    </alarms>
  </category>
  <category is-default="false" path="CATEGORY2">
    <description>Test category 2</description>
    <alarms>
      <FaultFamily>MF</FaultFamily>
      <FaultFamily>PS</FaultFamily>
    </alarms>
  </category>
</categories>
```

`Categories.xml` must have `categories` as document tag. The document is composed of an array of `category`. Each category is as described in the following table.

<i>category elements</i>	<i>Values</i>
description	The description of the category
alarms	The fault families of the alarms that must be associated to this category. It is composed of an array of <code>FaultFamily</code> elements each of which contains the FF of the triplets of the alarms to be published in the category

`category` has 2 mandatory attributes: `path` and `is-default`. `path` represents the name of the NC used by the category. In the example upon, you can see that `CATEGORY1` is used for the first category and `CATEGORY2` is used for the second one. Meaningful names should be used for the categories because the users select the categories understanding the types of the alarms that the categories will publish mainly by reading their names.

The boolean attribute `is-default` is also mandatory. In the definitions of the categories one category can be set as default. It means that whenever an alarm is received and it is not associated to any category (i.e. its FF is not present in any of the `alarms` element of the categories) then the default category will be used to publish that alarm. This avoid duplicating information and keeping the definition of categories short.

As we said before, if the ASC can't associate an alarm to one category then the alarm will not be published to any category and therefore will not be visible by client applications like the operator GUIs. Forgetting to associate an alarm to a category is a common mistake: it appear as the alarm is sent by a source but lost inside the ASC. by checking the logs you can easily understand what's going on because the ASC will warn that a log will not be published in any category¹⁵. The definition of a default category avoids losing alarms not associated to any categories, of course.

As we said for `fault-member-default`, the usage of a default category is very helpful avoiding mistakes and taking the definition of the CDB short. But on the other hand it should avoided because it breaks the concept of categories and the logical association of alarms to categories that at the end of the process means the creation of views for different types of alarm system users.

In the example upon, you can see that alarms for the families `Antenna` and `Mount` are published in `CATEGORY1`, while `PS` and `MF` are published in `CATEGORY2`. Being that `CATEGORY1` is also the default category, all the triplets having a FF not equal to `PS` and `MF`, will be published in that category.

There are no rules to associate alarms to categories. As a guideline we suggest that each ALMA subsystem publishes alarm in one category with a name resembling the name of

¹⁵ The ASC can only log a messages because the lack of association of an alarm to one category can be a desired behavior. On the other hand it can also be a mistake of the administrator and a presence of a log messages helps fixing the problem.

the subsystem itself. It will make easier identifying the entity generating the alarm and the responsible subsystem.

4.5 ReductionDefinitions

The ReductionDefinitions folder contains ReductionDefinitions.xml:

```
ReductionDefinitions
ReductionDefinitions.xml
```

The document type of the XML is reduction-definitions and contains 2 sections, links-to-create with the reduction-link and thresholds with the thresholds for the multiplicity reduction rules:

```
<reduction-definitions ...>
  <links-to-create>
    <reduction link...>
      ...
    </reduction-link>
    ....
  </links-to-create>
  <thresholds>
    <threshold>
      ...
    </threshold>
    ...
  </thresholds>
```

The configuration file describes the reduction rules (RR) representing the knowledge base used by the ASC to mask/hide alarms to the operators.

As we said before, there are two kinds of reduction rules, node and multiplicity. Each rule joins two alarms identified by their triplet. Each RR has a type, NODE or MULTIPLICITY.

The multiplicity reduction (MR) is used to reduce the number of alarms of the same kind presented to the operator. When the ASC receives n alarms of the same kind and n is greater than a given threshold then the ASC masks all the alarms and produce a new alarm for the user. It is important to remember that the alarm shown to the operator is a completely new alarm generated by the ASC and not by a source. The new alarm generated by the ASC must be defined in the CDB in order to present to the operator the right information.

As we said before, the MR is not activated by sending the same active alarm n times. It is instead activated when n alarms are active at the same time and such alarms are all part of the MR described in the database.

The node reduction (NR) aims to mask an alarm when it is known that a failure in a equipment is always triggered by the failure of another one. In this case the ASC shows the root cause of the problem, masking the other alarm.

Each RR joins 2 alarms defining their correlation. The ASC in turns joins more RR building a chain of reduction that culminates showing only the root cause of a cascade of alarms.

This is an example of an NR:

```
<reduction-link type="NODE">
  <parent>
    <alarm-definition fault-family="PS"
      fault-member="ALARM_SOURCE_PS" fault-code="1"/>
  </parent>
  <child>
    <alarm-definition fault-family="Mount"
      fault-member="ALARM_SOURCE_MOUNT" fault-code="1"/>
  </child>
</reduction-link>
```

As you can see, the type of the reduction is represented by the `type` attribute of the `reduction-link` tag. The RR is composed of a parent alarm and a child alarm. The meaning of the RR is that every time there is an alarm of type `<PS, ALARM_SOURCE_PS,1>` then there is also an alarm of type `<Mount, ALARM_SOURCE_MOUNT,1>` (i.e. whenever there is a failure of type 1 of the `ALARM_SOURCE_PS` then it triggers a failure of `ALARM_SOURCE_MOUNT`).

If the PS fails, then the MOUNT fails but the ASC knows that the failure of the MOUNT is triggered by the PS failure so it masks the second alarm: the operator sees only the alarm of the PS that is the root cause of this chain of failure.

Let's suppose that the failure of the PS, triggers the failure of the MOUNT that in turn triggers the failure of the ANTENNA. In this case the root cause of the chain of events is again the failure of the PS and we have to model the knowledge base of the ASC with this new correlation. In this case we only need to add the following NR:

```
<reduction-link type="NODE">
  <parent>
    <alarm-definition fault-family="Mount"
      fault-member="ALARM_SOURCE_MOUNT" fault-code="1"/>
  </parent>
  <child>
    <alarm-definition fault-family="Antenna"
      fault-member="ALARM_SOURCE_ANTENNA" fault-code="1"/>
  </child>
</reduction-link>
```

The case of the MR is analogous: the type of the RR must be set to `MULTIPLICITY` instead of `NODE`. The parent alarm is the alarm generated by the ASC when the number

of active alarms is greater than threshold. The child alarm is the alarm generated by a component.

The `thresholds` section of the XML defines the thresholds for the MR. Each threshold is defined by an integer, the threshold, and the triplet of the alarm generated by the ASC (i.e. the triplet of the parent in the MR).

The alarms that are counted as part of a MR are all the alarms defined in a MR entry having the same alarm (generated by the ASC) as parent.

In the following example, you can see two alarms that are part of the same MR: `<MF, MULTIPLE_MF_FAILURES, 0>` and `<MF, MULTIPLE_MF_FAILURES, 1>`. When both of them are active (remember that the threshold is 2), the ASC generates an alarm of type `<MF, ALARM_MULTIPLE_MF_FAILURES, 5>`.

The following is an example of a definition of such a MR:

```
...
<reduction-link type="MULTIPLICITY">
  <parent>
    <alarm-definition fault-family="MF"
      fault-member="MULTIPLE_MF_FAILURES" fault-code="5"/>
  </parent>
  <child>
    <alarm-definition fault-family="MF"
      fault-member="ALARM_SOURCE_MF" fault-code="0"/>
  </child>
</reduction-link>
<reduction-link type="MULTIPLICITY">
  <parent>
    <alarm-definition fault-family="MF"
      fault-member="MULTIPLE_MF_FAILURES" fault-code="5"/>
  </parent>
  <child>
    <alarm-definition fault-family="AlarmSource"
      fault-member="ALARM_SOURCE_MF" fault-code="1"/>
  </child>
</reduction-link>
....
<thresholds>
  <threshold value="2">
    <alarm-definition
      fault-family="MF"
      fault-member="MULTIPLE_MF_FAILURES"
      fault-code="1"/>
  </threshold>
  ...
</thresholds>
```

To make the MR example short and readable, we have published alarms having the same FFs and FMs but with different FCs. While defining MR rules you can define triplets of any kind in the child elements.

The triplet generated by the ASC can also be of any kind but it needs to be defined in the CDB in order to be visible to the operators (i.e. its definition must be present in `AlarmDefinition`, and its family must be associated to a category in `Categories.xml` as you can see browsing the CDB of the demo example).

The alarms that are not part of any RR are not masked by the ASC and they will appear in the operator GUI.

There are no defaults available for the reduction rules definitions: the application of such rules make some alarms invisible in the operator GUIs and we do not want this happens without the explicit control of the administrator.

5 The alarm GUI

ACS provide a java SWING GUI showing the alarms in for of a table. Such a GUI runs as OMC plugin and as a stand alone application by launching the `alarmPanel` command.

When the panel connects to the ACS it gets the list of the active alarms from the ASC and shows them in the table. You can then start the panel at any time and have immediately visible the active alarms regardless if they have been issued before launching the panel. Each alarm has a color depending on the priority of the alarm and its state. If the alarm is inactive, it is shown in green. If it is active, the color depends on its category: red for priority 0, orange for 1, yellow for 2 and light yellow for priority 3. Inactive alarms are displayed in green but not removed from the table by default. To remove an alarm the user has to press the right mouse button over an alarm and select the Acknowledge popup menu item. It is important to remark that active alarms can't be removed by the table.

The actual version of the panel, automatically connects to all the existing categories without asking the user for a confirmation.



Figure 1: the alarmPanel

The panel is divided in three main areas: the toolbar in the upper side, the alarm table in the center and the status bar at the bottom

The left side of the status area shows 5 colored widgets with a number in the middle. Each widget shows a summary of the alarms in the table of a given priority, the number is the number of active alarms of that priority. An alarm has one of the four priorities and therefore there are four colored widgets from red, top priority to light yellow, lowest priority. The last widget, the green one, reports the number of inactive alarms in the table. Those widgets are refreshed approximately every 2 seconds independently of the flow of the alarms i.e. it may happen that the numbers shown in the widgets and the number of alarms in the table are misaligned for a short time.

The right side of the status bar shows an icon that is green when the panel is connected to the ASC and gray when it is trying to connect to the ACS.

The table in the middle of the GUI shows the alarms, one for each line of the table. The alarms are colored depending on their priority. Initially the alarms are sorted by putting a new alarm on top. If an alarm becomes inactive, it appears colored in green but does not

change its position in the table. Vice versa, if an alarm is inactive and becomes active again, it is moved on top of the table.

The user can change the sorting of the table by pressing the column headers. The table sorts the alarm by 2 level remembering the two last selected columns. For example if you want to sort the table by component and then by priority you have to press the priority header and then the component header. Table columns can be moved by selecting their header and dragging to the desired position.

A popup menu is shown by pressing over a row with three items:

- Acknowledge: as said before it is used to remove inactive alarms from the table; this item is disabled if the alarm below the mouse pointer is active
- Save: save a text file containing the alarm below the mouse pointer
- To clipboard: copy the alarm in the clipboard

By default the time, the component, the priority, the description and the cause of each alarm are displayed. But it is possible to customize the fields of the alarm displayed by pressing the right mouse button over the table header and selecting the switches to activate/deactivate as shown by the following picture.

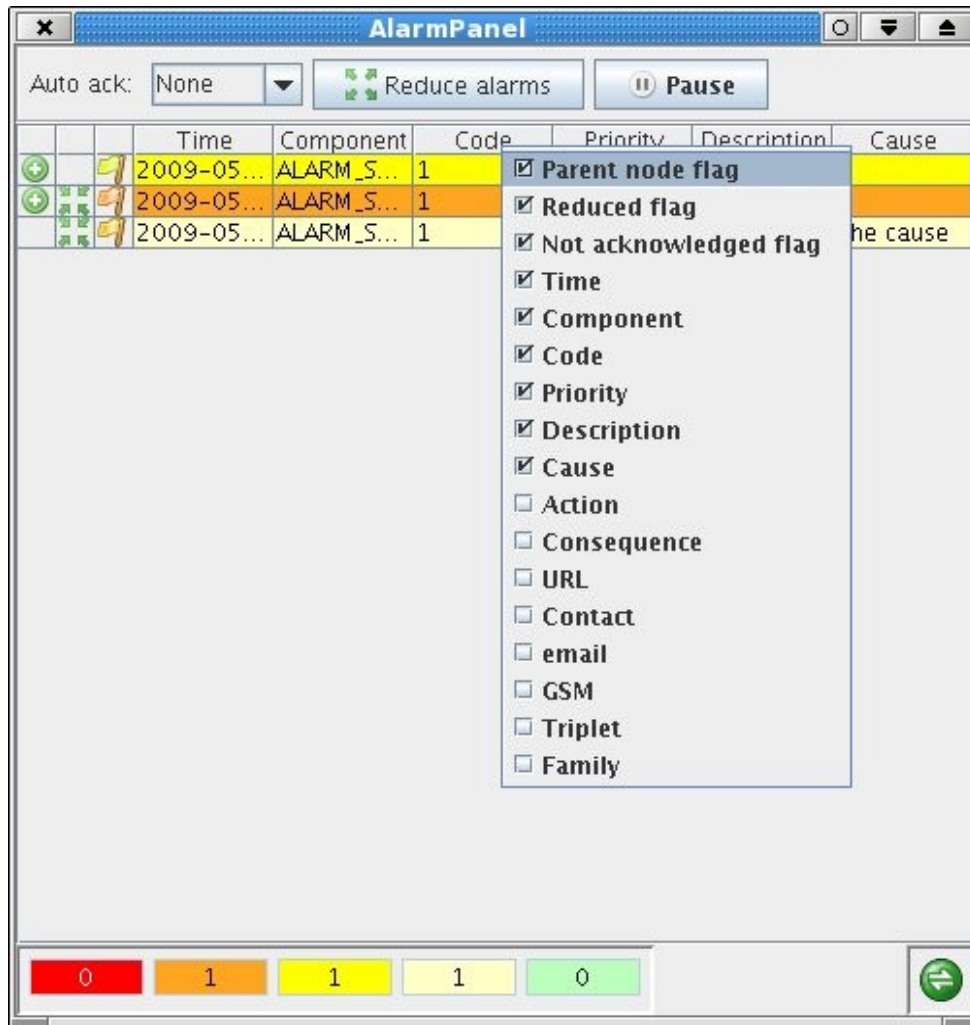


Figure 2: the table of alarms without reduction rules

The figure upon shows the alarms generated by the demo example without activating the reduction rules. The first two columns shows is an alarm is involved in a reduction rule. The first column has a green icon with a plus if a reduction rule that alarm hides another alarm. The second column shows a green icon with four little arrows pointing to the center if the alarm is hidden by a reduction rule.

Note that an alarm can cause the hiding of another alarm and be hidden at the same time.

If the reduction rules have been configured in the CDB, the alarm system marks the alarms involved in the reduction rules with a special set of flags. The GUI reads those flags and is able to hide the unneeded alarms showing only the root cause of a problem. This is very important because it can dramatically reduce the number of alarms in the table and enhance the readability of the table as well as the understanding of the problem by the operators.

In the previous example in fact, if we activate the reduction rules we can immediately see that the cause of the chain of alarms was a failure in the power supply so we know where to look to fix the failure.

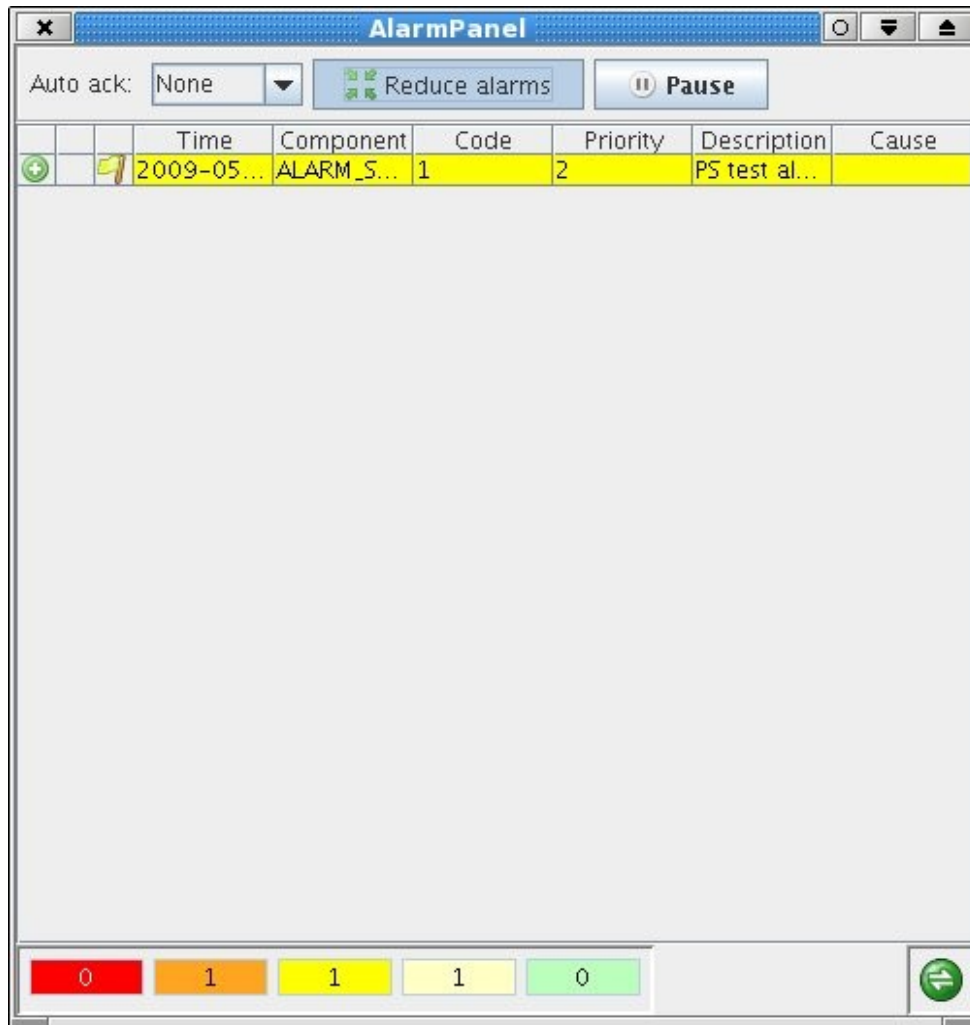


Figure 3: the panel with the reduction rules activated

It is often very useful to investigate which alarms are involved in a reduction rule. For example if we have only one alarm visible like in the previous figure it could be useful to see the full chain of alarms involved by that reduction rule. It can be done by moving the mouse pointer over the alarm of the table, pressing the right mouse button and select the *show reduction chain* item of the popup menu.

The reduction chain of an alarm is shown by means of a dialog that shows the alarm in a flat view, the table or in a tree view better showing the relationship between involved alarms.

	Time	Component	Code	Priority	Description	Cause
+	2009-0...	ALARM_S...	1	2	PS test al...	
+	2009-0...	ALARM_S...	1	1	Mount test	
+	2009-0...	ALARM_S...	1	3	Test ante...	The cause

Figure 4: the reduction chain of the PS alarm

The table with the reduction chain is aimed to show the relationship between alarm and is simpler than the table of alarms in the main window. For example it is not possible to select the columns to show but it is possible to sort the table by pressing over the column headers.

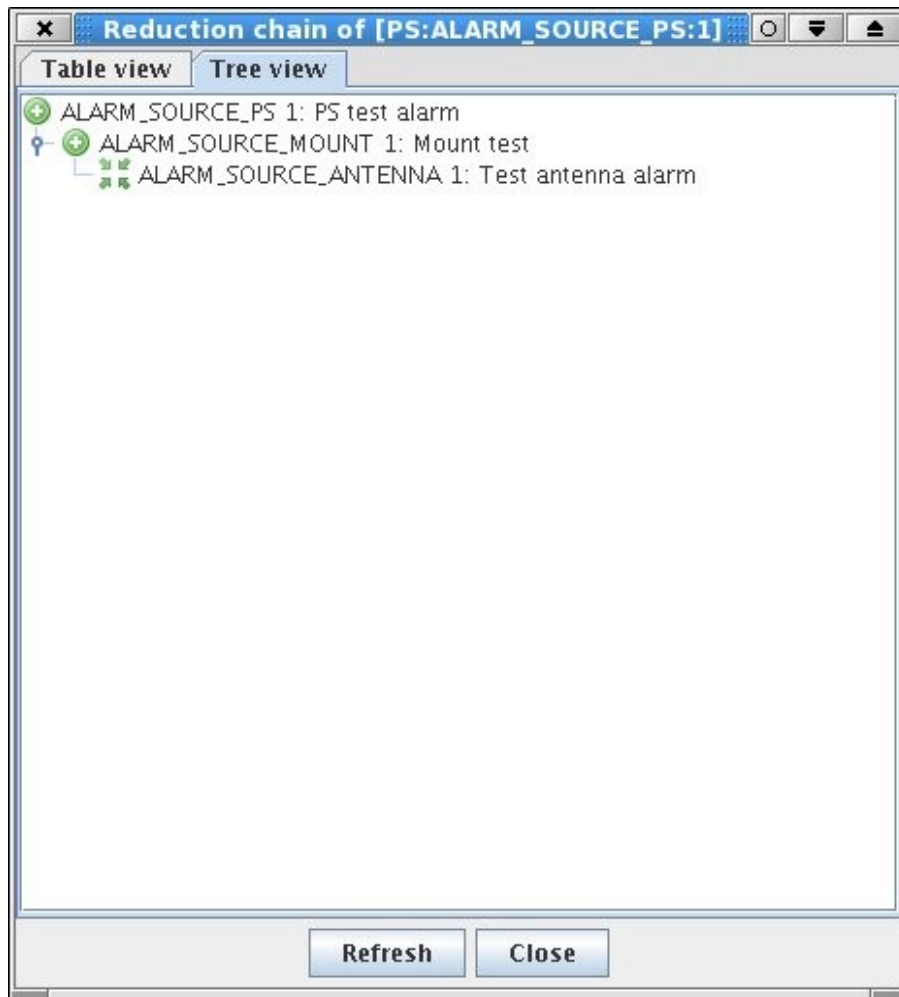


Figure 5: the reduction chain of the PS alarm as a tree

The tree view is another way of viewing the same reduction chain that better shows who is hiding what. As a general suggestion, the tree view fits better for NR and the table view for MR.

When an alarm becomes inactive, i.e. the failure has been fixed, its color turns to green but the alarm is not removed from the main table of alarms until the user acknowledges the alarm by pressing the right mouse button and selecting the appropriate menu item.

The toolbar has a menu to select the auto-acknowledge level (disabled as default). By setting the auto-acknowledge level all the alarms having the selected priority, or a lower one, are transparently removed by the table when they become inactive. It is not possible to set the auto-acknowledge level for priority 0 alarms that must be explicitly acknowledged by the user by pressing the right mouse button. Auto acknowledgement allows to reduce the number of visible alarms but does not delete the history of what happened to the system and therefore should be avoided or used very carefully.

Note that at the present, the acknowledgment of an alarm is done locally simply removing the entry from the table. In a future release the acknowledgment will be propagated to all the opened panels.

The max number of alarms displayed by the table is limited to 20000. If the table is full and a new alarm arrives, the the oldest is removed.

The pause button freezes the refresh of the table of alarms enhancing the readability of the table.

6 The CERN operator GUI

The CERN alarm system provides a netbeans GUI to show the alarms. Unfortunately such a GUI does not run inside the OMC and therefore we implemented a SWING GUI, the alarmPanel. The netbeans GUI is still part of ACS distribution and can be run as a stand alone application.

The ACS swing panel is the default operator panel and is automaticaly built by ACS. If you wish to try the CERN GUI, it must be manually built: it is in the module ACS/LGPL/CommonSoftware/NetbeansACS.

The CERN operator GUI is started with the `acsalarmgui` command: a script that reads the actual configuration and starts the netbeans application. For it to work well you need ACS correctly installed, and the ASC running.

`acsalarmgui` is a python script that accepts only one parameter, the corbaloc of the manager; it sets up all the parameters needed by the GUI before starting it. It looks for the manager reference from the command line, the `MANAGER_REFERENCE` environment variable and the local host. The `ACS_INSTANCE` is read from the environment variable or set to 0 if missing.

The script takes care of preparing the classpath, copying and installing the last version of the jars in the current directories as expected by netbeans. The packages are searched in the `INTROOT`, `INTLIST` and `ACSROOT` as usual.

The GUI is written in netbeans but a new SWING version is available at CERN and will be ported as soon as possible into ACS.

At the present, the user configurations are not stored on the database so each time the GUI starts, the user needs to setup the configuration. After login as test (the only available user), a configuration dialog is shown. The user should select the categories he wants to listen to. This is done by selecting the `ROOT` category and the adding all its subcategories. Another important setting is the activation of the Reduction in the behavior tab.

When finished setting up the configuration, the user has to approve the changes and finally close the configuration window.

At this point the GUI is connected to the AS, as shown by the green icon in the right bottom side of the window.

The main table in the center of the windows shows the incoming alarm.

Each new alarm is shown with a “N” in the left side of the line. When the user selects the alarm, the N disappear and is replaced by the date when the alarm was received. The purpose of the N is to show the user that an alarm is new and has not yet been seen (selected).

If the reduction is disabled, the table will show all the alarms published by the ASC. If the reduction is active, then only the root cause of a chain of alarms is shown and all the other alarms are hidden reducing drastically the amount of alarms shown in the table.

The alarms do not disappeared from the GUI even if the abnormal situation has been fixed. This is to force the user to explicitly acknowledge each alarm by pressing a right mouse button over the alarm.

We suggest to play a little bit with the GUI to understand its functioning and options. However, not all the functionalities of the original LASER GUI are available in this version. No further work is foreseen in this interface because all the efforts will be to adopt the new SWING version.

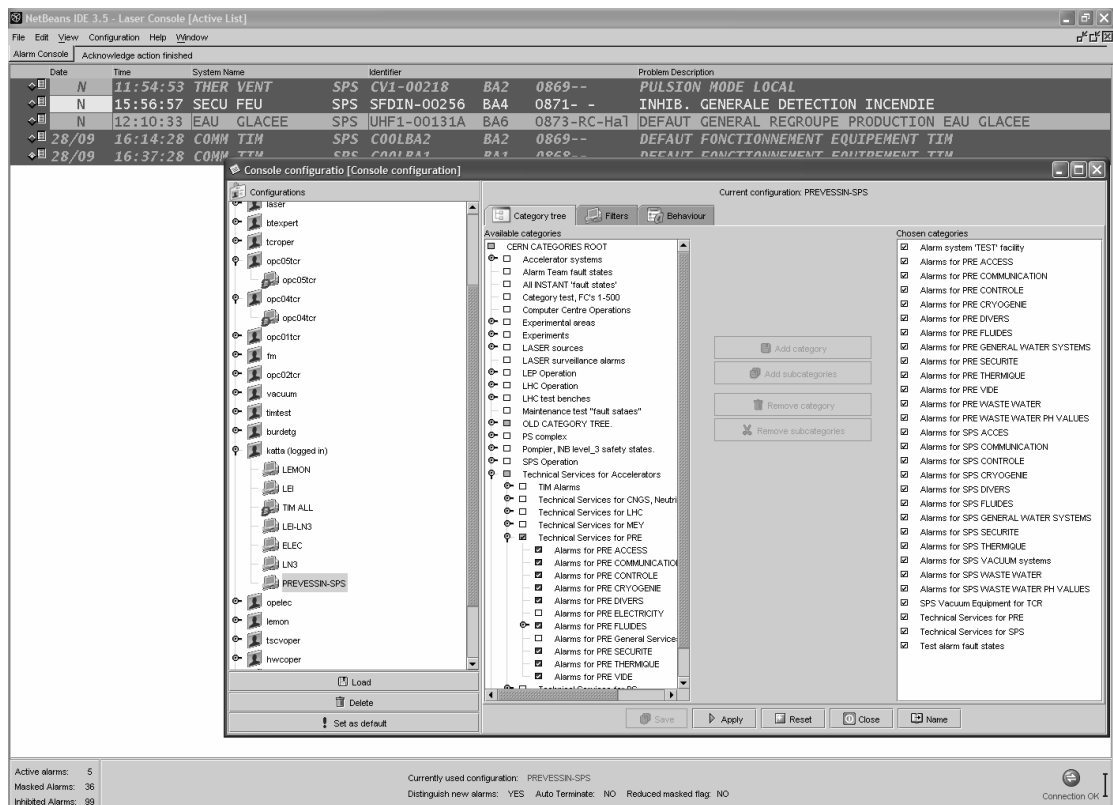


Figure 6: A snapshot of the operator GUI from the LHC control room.

The picture above shows a black and white screenshot of the GUI used at CERN. The inner window is the user configuration dialog.

The left side of the configuration window, shows the user configurations currently available. The setup is in the center of the window.

In the left side, all the available categories are shown. If you are using the test CDB, you should see the root category and two subcategories¹⁶: current and source. You should select the ROOT and press the add subcategories button: doing that you will see the selected categories shown in the right part of the window as shown in the picture below.

¹⁶It might be that you have to wait a little bit before seeing the current and source subcategories below root in the Availbale categories list.

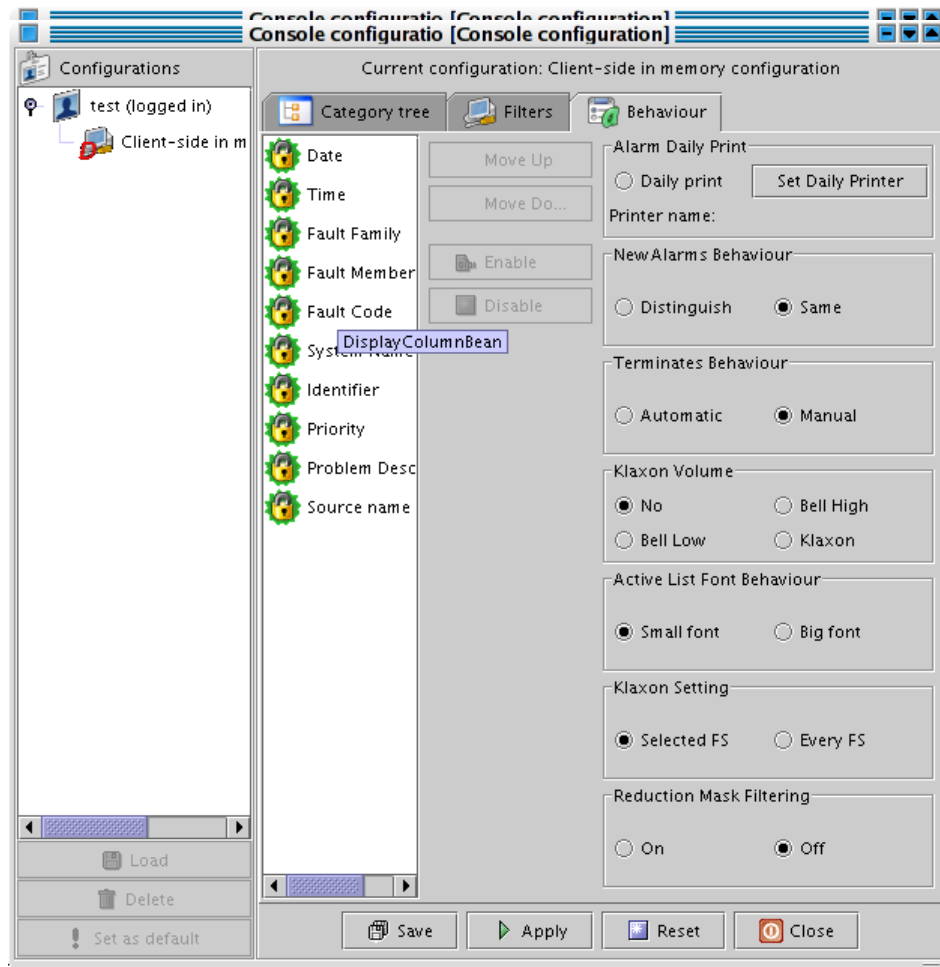


Figure 8: The console configuration dialog: behavior definition.

The behavior tab presents more options. The last one, reduction Mask Filtering enables and disables the reduction of alarms. This dialog is shown below.

Once the configuration is ready, press the apply button and then the close button.

The main window is composed by a central table where all the alarms are shown, one per line. The picture below shows the GUI without reduction with all the alarms generated by the demo. The snapshot is taken with the test CDB and without reduction. If the reduction are active, the PS would be the only alarm shown because it is the root cause of the chain of alarms as defined in the CDB.

The alarms are shown with different colors depending by their priority.

The green icon in the right bottom shows the status of the connection with the ASC. The antenna and PS alarms are new in the meaning that the user has not yet selected them. The Antenna instead has been selected and the N has been replaced by the actual date.

The alarms are all active. If the user executes the terminate method of the PS, all the alarms become inactive and their color in the GUI switches to green. Inactive alarms are not removed from the table because the user must explicitly acknowledge the abnormal situation.

This is done by pressing the right mouse button of the alarm and selecting the Acknowledge item of the popup menu: a new dialog is shown where the user has to write his comment into. Only at this point the alarm is removed from the main window.

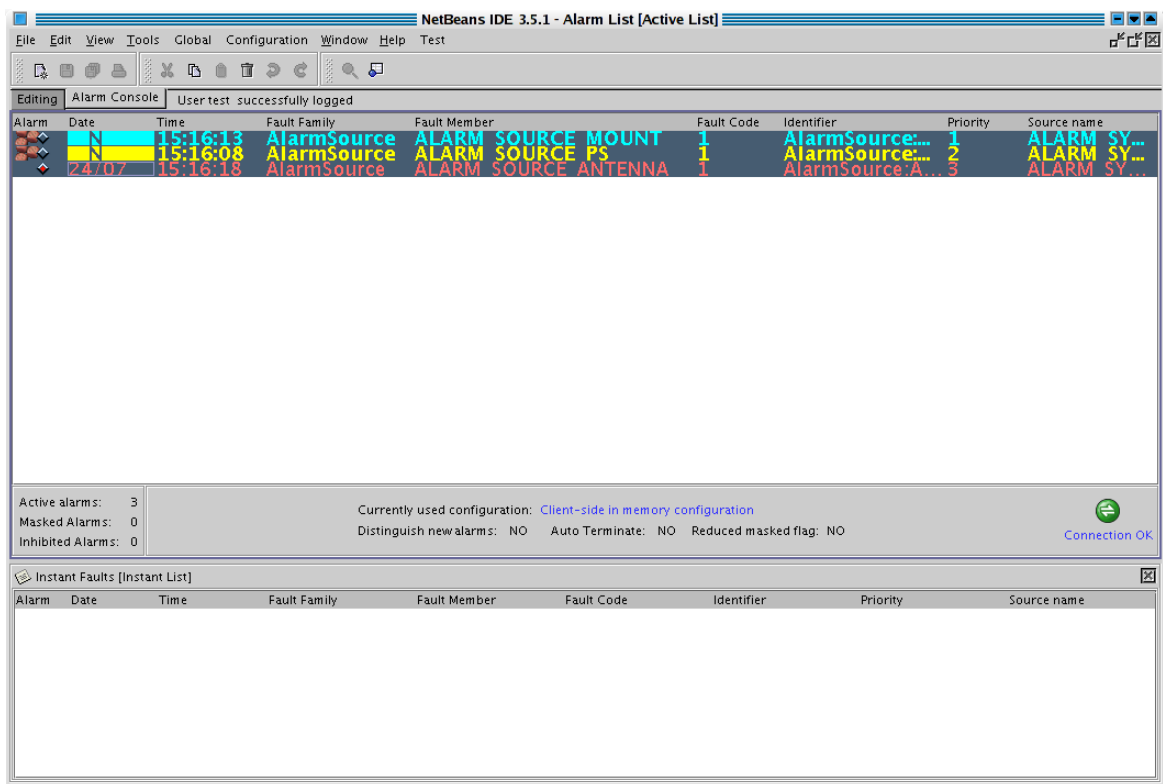


Figure 9: The main window of the operator GUI with the alarm generated by the demo and the reduction rules disabled.

As we said before, a new version of the GUI, written with java SWING library is ready at CERN and we do not plan to work further on this version. The look and the feel as well as the functioning of the two GUIs is very similar.

Not all the functionalities of the original GUI have been ported in ACS so it could happen that you try to do something not yet implemented or not working. We kindly suggest you to play with the GUI keeping in mind that every missing functionality or not working feature can't cause further malfunctioning in the system.

7 ACS implementation

ACS specific code has been developed to adapt CERN alarm system to ACS or to extend the basic CERN implementation to fit ALMA needs.

To better understand the following discussion, it is better to refer to the following figure.

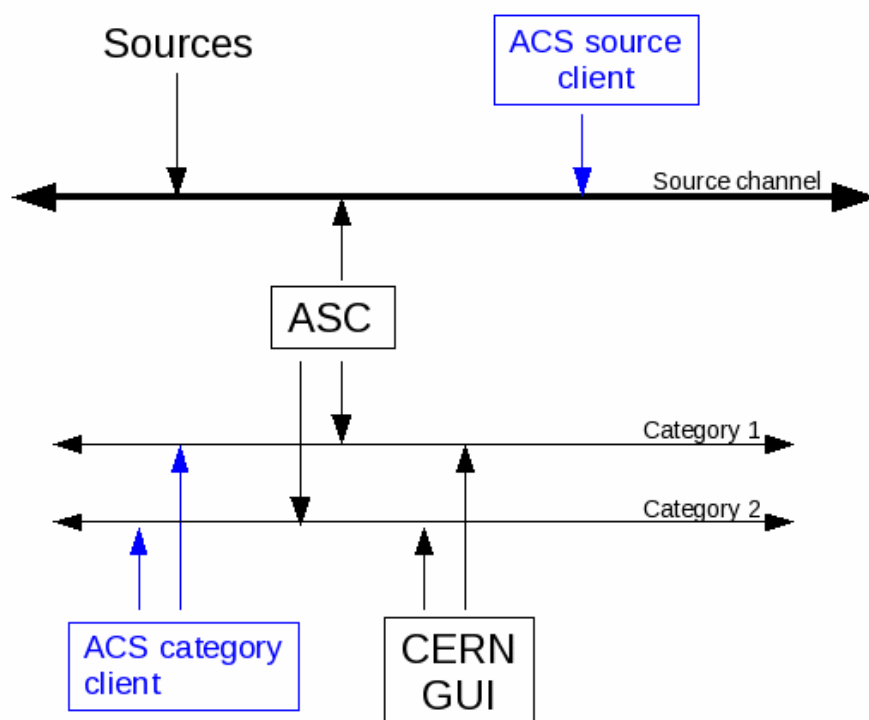


Figure 10: CERN architecture and ACS clients

Black lines and boxes represent the CERN part of the alarm system. The sources publishes the alarms in the source channel where the ASC listens to. For each received alarm, the ASC publishes its enriched view in one or more category channel where the CERN operator GUI listens to alarms to show in the table.

ACS adds the blues lines and boxes, i.e. clients to listens to sources and categories alarms. The graph shows the different view of the alarms of the two ACS clients¹⁷.

ACS source and category clients are in turn used by the alarmSourcePanel and the alarmPanel to display alarms .

ACS category client is an API to access alarms published by the ASC, i.e. the same alarms the CERN operator GUI shows in the table. In this context, ACS category client is an helper class to easily access categories from inside ACS code.

The ACS source client is instead something not present in CERN alarm system where nobody is allowed to listen to source alarms besides the ASC.

7.1 Alarm source client

SourceClient is a java alarm source client i.e. it connects the the source channel and listen to the alarms published by the sources. The client is used by the alarmSourcePanel.

The following example shows the usage of the source client:

```
private ContainerServices contSvc;  
private SourceClient sourceClient= new SourceClient(contSvc);  
sourceClient.addAlarmListener(this);  
sourceClient.connect();  
...  
public void faultStateReceived(FaultState faultState) {  
    System.out.println(faultState.getFamily());  
}
```

The client is initialized passing an instance of ContainerServices. The listener to the alarms must then be added to the source client that will start sending alarms after the call to connect().

Alarms are received in the faultStateReceived method of the listener (in the example above, the name of the family of each received alarm is written in the stdout).

This class can be of help when writing tests of the alarms sent by sources.

7.2 Alarm category client

CategoryClient is a category client allowing to listen to alarms published by the ASC into categories. This client is used by the alarmPanel.

The following shows the usage of CategoryClient:

```
import alma.acs.container.ContainerServices;
```

¹⁷ Both the clients are in ACSLaser/alarm-clients.

```

import alma.alarmsystem.clients.CategoryClient;
import cern.laser.client.data.Alarm;
import cern.laser.client.LaserException.LaserSelectionException;

...
private ContainerServices contSvc;
private CategoryClient categoryClient;
...
categoryClient=new CategoryClient(contSvc);
categoryClient.connect(this);
...
public void onAlarm(Alarm alarm) {
    System.out.println(alarm.toString());
}
...
public void onException(LaserSelectionException e) {
    ...
}

```

The client is built passing the ContainerServices.

The call to the connect triggers a sequence of actions:

- the client connects to the alarm system and gets the list of all the available categories
- the listener is connected to the alarm system to be notified about the arrival of alarms
- the category client gets all the active alarms i.e. all the alarms active, even those activated before the client connects to the alarm system
- all the active alarms are sent to the listener

A not null listener to receive the alarms published by the ASC must be set as parameter to the connect. The CategoryClient sends alarms to the listener through onAlarm, as defined by the AlarmSelectionListener interface.

By default, the client connects to all the available categories, as shown in the example. However, there is an overloaded connect that accepts an array of Category to connect to listen to alarms:

```

public void connect(AlarmSelectionListener listener, Category[] categories)

```

7.3 alarmSourcePanel

The alarmSourcePanel is a little dialog showing alarms published by sources i.e. alarms before they are processed by the ASC.

The panel, that needs to be started against a running ACS session, connects to the source channel and shows the alarms published by the sources inside a table. Note that the alarms shown in this table are the same alarms received by the ASC.

Family	Member	Code	Activator
Antenna	ALARM_SOURCE...	1	ACTIVE
Mount	ALARM_SOURCE...	1	ACTIVE
PS	ALARM_SOURCE...	1	ACTIVE

Figure 11: A snapshot of the alarm source panel

This panel is very useful for debugging when the user wants to monitor the alarms published by sources. The table shows the triplet and the activator (the ACTIVE/INACTIVE status) of each alarm.

7.4 BACI properties

BACI properties have been modified in order to send an alarm when their value becomes lower than the minimum or higher than the maximum. The min and max possible values of a property are defined in the CDB.

This feature must be explicitly enabled by the user changing the value of *alarm_timer_trig* in the CDB entry of the property¹⁸. The default value for this field, defined in the schema, is 0 meaning that the alarm is disabled.

If this value is set to a number greater than 0 then the changes to the value of the property are continuously read and compared to its min and max in order to detect if it passes the limits.

Besides the *alarm_timer_trig*, the methods `startPropertiesMonitoring()` and `stopPropertiesMonitoring()` of `CharacteristicComponentImpl` allow the developers to enable/disable the monitoring of the properties.

All the alarms published for the BACI properties have the following triplet:

```
< FF="BACIProperty", FM=property_name, FC=1 >
```

¹⁸ Please, refer to the BACI documentation for further details about the CDB entry of a property.

The FS of each BACI property can be ACTIVE or INACTIVE depending on whether its value is in the proper range or not.

As discussed in the CDB section, the sources can publish alarms even if they are not defined in the database. On the other hand, the ASC discards all the alarms it receives that do not match with an entry in the CDB.

In other words, it means that there is nothing to configure in order for a BACI property to send alarms when its value is out of the allowed range. However you must configure the Alarms branch of the CDB if you want to have these alarms processed by the ASC and visible in the GUI, that is normally the desired behavior. Usage of default members is very helpful defining the CDB for BACI properties.