Dynamic Error Recovery in the ATLAS TDAQ System

John Erik Sloper, *Student Member, IEEE*, Giovanna Lehmann Miotto, and Evor Hines

***Abstract*—This paper describes the new dynamic recovery mechanisms in the ATLAS Trigger and DataAcQuisition (TDAQ) system. The purpose of the new recovery mechanism is to minimize the impact certain errors and failures have on the system. The new recovery mechanisms are capable of analyzing and recovering from a variety of errors, both software and hardware, without stopping the data-gathering operations. An expert system is incorporated to perform the analysis of the errors and to decide what measures are needed. Due to the wide array of sub-systems there is also a need to optimize the way similar errors are handled for the different sub-systems.**

The main focus of the paper is to consider the design and implementation of the new recovery mechanisms and how expert knowledge is gathered from the different sub-systems and implemented in the recovery procedures.

***Index Terms*—ATLAS, error recovery, expert system, TDAQ.**

I. Introduction

THE ATLAS High-Level Trigger and DataAcQuisition (TDAQ) system is a large heterogeneous system consisting of a wide variety of software and hardware components. The final production system will consist of approximately 3000 nodes running more than 20 000 processes. The system is divided into a number of sub-systems each performing a well defined task. A detailed description of the TDAQ system can be found in [1].

II. Background and Motivation

The TDAQ system has been under development for more than a decade. Even so, the development of an advanced Error Recovery (ER) system has not been a priority. Previous error recovery systems have been simple and, thus, were only able to handle trivial cases such as restarting or ignoring a dead process or putting the system into an error state. As we approach the start-up of the experiment, larger and larger system configurations are being used. Ideally, the system should gather data from the detector over long periods of time (several days) without interruption. Due to the very large time frames and volumes of data to be collected, errors will most certainly occur, and it is important that it is possible to recover from these errors without the need to restart the particular stage of the experiment. It should also be possible to perform this recovery with the need for as little user interaction as possible. The various sub-systems will have their specific requirements for handling errors and errors might have different impact, depending on the state of the sub-system or the system as a whole.

Manuscript received May 10, 2007; revised November 15, 2007.

J. E. Sloper is with CERN, CH-1211 Geneva, Switzerland, on leave from the University of Warwick, Coventry, CV4 7AL, U.K. (e-mail: john.erik.sloper@cern.ch)

E. Hines is with the University of Warwick, Coventry, CV4 7AL, U.K.

G. L. Miotto is with CERN, CH-1211 Geneva, Switzerland.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNS.2007.913472

III. Requirements

There are a number of main requirements that have guided the general design of the ER system.

- **Error handling**: The ER should be able to react to errors reported in the system, analyze them and take appropriate actions to ensure that the system returns to or stays in an operational state.
- **Customizable**: It must be possible to ensure that different behavior will take place in response to similar, or indeed identical, problems arising in different sub-systems. For example, a dying application within one of the computing farms might have no overall effect on the system other than a slight reduction in computing power. However, if an application in the ReadOut System (ROS) dies, this will have considerable impact on several other sub-systems.
- **Configurable**: It must be possible to easily change the behavior of the recovery system. The TDAQ system is under development, and it must therefore, when completed, be able to accommodate for new components. Even in the final system, there will be a relatively large number of different configurations to run. It must, therefore, be possible to configure the ER system to accommodate any needs that may arise.
- **Abstraction**: The ER should be such that any parts can be replaced without the need to change any other components. For example, applications should be able to report errors through a fixed interface without knowing anything about the mechanisms of the recovery system. This will also help to minimize the need for change in user implementation. This is very important because the ER system is being introduced relatively late in the development of the TDAQ system.
- **Performance**: The ER should be able to analyze the errors and reach a decision within reasonable time span, usually on the order of seconds. The longer these decisions take, the longer the TDAQ system will remain in an error state, causing a loss off data taking capability.

IV. Design and Implementation

The ER system is closely related to the TDAQ RunControl system, and naturally, design and implementation choices are guided by this. The following sections will first give a brief

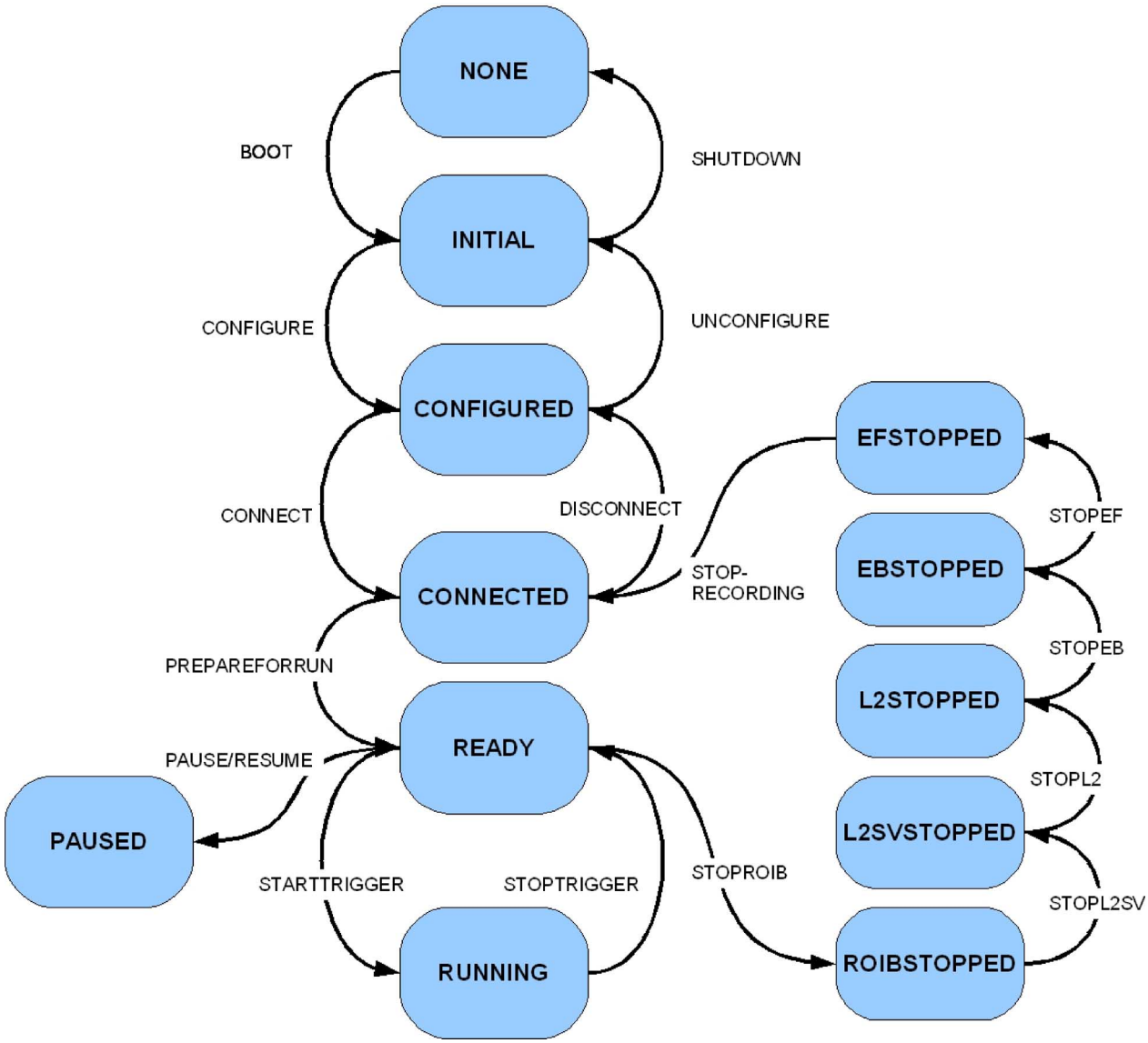

Fig. 1. TDAQ FSM.

overview of the TDAQ system, including some of the most important components. We will then move on to consider the Run-Control system in detail before discussing specific design and implementation choices for the ER system.

A. TDAQ System Structure

Applications in the TDAQ system are organized in a tree structured manner. In addition, there are a number of services that are provided which use a classic client-server model. Communication is realized using a dedicated Inter Process Communication (IPC) package based on CORBA [2]. The most important services in the context of the recovery system are the Message Reporting System (MRS) and the Error Reporting Service (ERS). The MRS is a service for passing messages between different applications using a subscription-notification model. The ERS provides several services, including a common format and a fixed range of severity levels for all errors reported in the TDAQ system. The ERS relies on the MRS in order to pass error messages between different applications.

B. Configuration Database

A common database stores the configuration of the TDAQ system. It contains everything from command line parameters to the overall organization of both hardware and software connections. The database defines a set of segment objects, typically representing a subsystem or a collection of applications with similar functionality, e.g., a set of Readout modules. Each segment contains a set of applications, resources, and other segments. The configuration database is available to all applications through a database server.

C. TDAQ Runcontrol

The TDAQ RunControl system is responsible for distributing commands from the operator(s) throughout the system. It starts, stops, and monitors all applications in the TDAQ system and ensures that the system is in a coherent state. To synchronize operations Finite State Machine (FSM) principles are used. Fig. 1 shows the FSM used for the TDAQ system. In addition to the

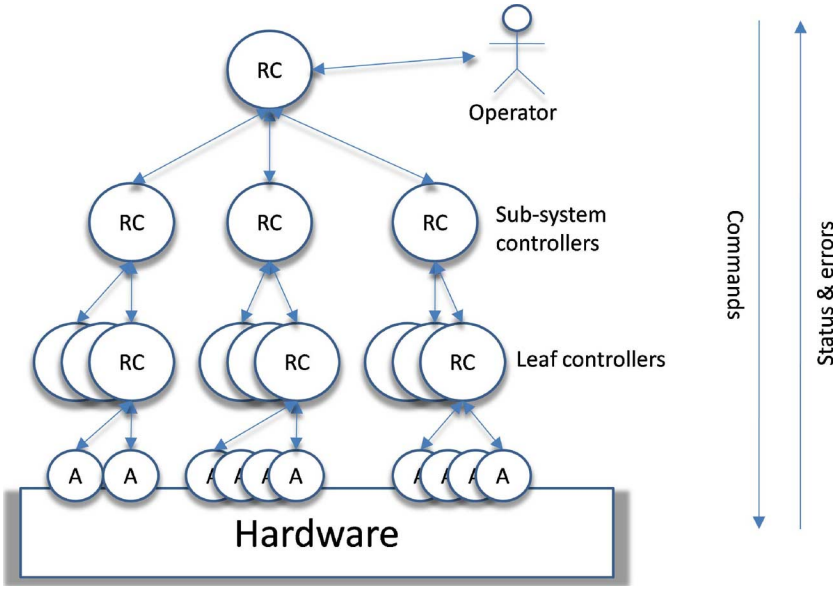

Fig. 2. TDAQ control tree.

states shown, it can also go into an error state, indicating that the application cannot continue its function.

The RunControl is constructed using the configuration database with controllers arranged in a tree structure, where each controller is responsible for a segment. Normally, commands are only sent to the topmost controller and are then propagated throughout the control tree. Interaction with the RunControl is performed through a graphical interface which, among other things, displays the RunControl tree, including the current state and any errors. Fig. 2 shows the logical layout of the RunControl.

D. Design

There are two main parts of the dynamic error recovery system: a *local* unit and a *global* unit. The local unit is integrated with each controller in the control tree. It has a complete "map" of the applications within its controller's segment. Any changes in the applications status are reported to the ER system by the controller. The main goal of the local recovery unit is to handle errors that can be dealt with at a segment level, that is, errors that do not have an immediate effect on the rest of the system. The local recovery unit receives information directly from the with which controller it is integrated. It will then analyze any problems, taking into account information such as the configuration, system state, other errors, and so on. It can set the error state of the controller, but will also be able to perform more advanced actions such as restarting applications or notifying other applications. In addition to receiving errors directly from the controller, it can also receive errors directly from applications. All errors gathered by the local unit will be reported to the global unit, including information such as whether the action has been taken and if the problem has been solved or not.

The global unit handles errors that have a system-wide impact, where applications in different segments will be affected by the recovery actions. It should also be able to take actions based on information received from the local units. The server keeps track of all errors in the system, including the ones reported from the local units. This allows for more general decisions to be made, based on parameters such as frequency of errors within a segment and/or the system as a whole.

Both the global and local unit has access to the configuration database and uses it to build maps of the connections in the system, read default recovery actions whose tests are associated with the different applications, etc.

In addition to handling errors from the TDAQ system, the ER must also be able to handle errors from systems outside of TDAQ such as the Detector Control System (DCS), networking, farm monitoring tools, and so on. This is realized by integrating a proxy application into the control tree and passing error messages through the proxy. A detailed description of this interaction in the DCS case can be found in [3].

The ER system is designed to interface with related components such as the Diagnostics and Verification System (DVS) [4]. This allows it to actively test components in the system. This is especially useful in cases where the actual fault is not immediately apparent. For example, if an application is reported not to be responding, the recovery system can test the network connections or the host of that application to see if the error is in fact a hardware problem. There are well-defined tests for different hardware objects and applications that can be used by the recovery system to properly identify the problem.

Due to the distributed nature of the system, it is, in most cases, not practical to perform a synchronous recovery. Even though the system does support synchronous communication between applications, the ER is designed to perform the recovery in an asynchronous manner. There are several reasons for this. First of all, to ensure that the abstraction requirement is fulfilled, it is better not to have a direct connection between applications that are reporting errors and the ER system. Also, due to time constraints, it might not be practical for some applications to wait for an answer from the ER system before continuing its

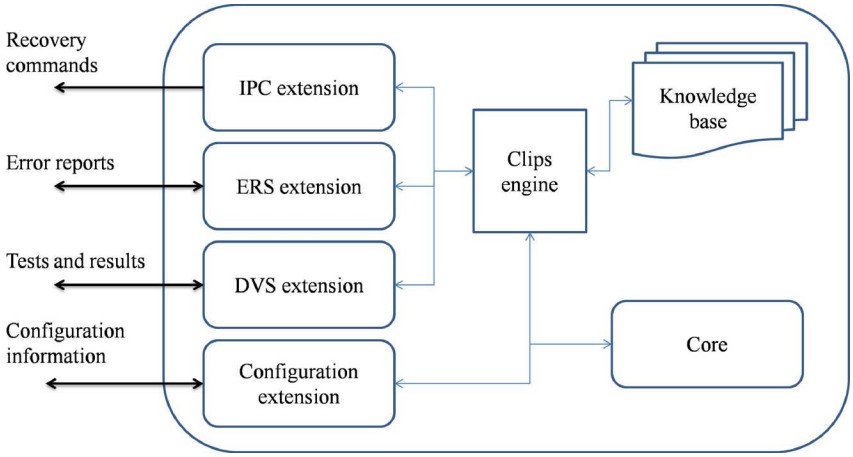

Fig. 3. General structure of both local and global ER units.

operation. For example, an application might report that another application is not responding, but it may still be able to continue its own operation.

E. Implementation

The two parts of the system are implemented in a similar way. Fig. 3 shows a diagram describing the building blocks of both the local and the global ER units. The main difference between them is that the local unit is directly integrated with a controller and can communicate directly with it. The global unit, on the other hand, is a standalone server that is completely outside the control tree. The global ER server also has a simple interface both for reporting errors to it (used by the local recovery systems) and for retrieving information about errors that have been reported and had actions taken. This is mainly used for monitoring purposes but can be a useful tool for a human expert as a help in identifying problems in the system.

Both the local and the global ER units rely on a rule-based expert system, which is also known as a knowledge-based system, to analyze errors and decide on appropriate recovery actions. The main advantages of a rule-based expert system are that it is simple to implement in the first place and that changes and additions can be easily made as the need arises. It is very difficult to predict *a priori* all the different errors that might occur and what appropriate actions should be taken. It is therefore very important that the expert system can be easily changed and customized as more data is gathered and a better understanding of the system is gained through experience. The main disadvantage with using a rule-based system is that it can be difficult to manage such systems as the size of the knowledge base increases. Experience suggests that when the number of rules grows, it becomes increasingly difficult to keep track of the impact each change will have on the system. This not withstanding, we find that with careful implementation, this problem can be avoided, and the flexibility of the approach makes it a good choice for our system.

The expert system is being implemented using CLIPS [5]. The CLIPS technology was chosen for several reasons. First of all, it has been used for many years in the ATLAS experiments, both in the controllers [6] and in the DVS framework, and therefore, there exists considerable experience among the developers.

```
if
system state is running, and
application App1 status is absent, and
application App1 has supervisor S1, and
application App1 membership in
then
notify S1 ignore App1
notify controller ignore App1
set membership App1 out
```

Fig. 4. Sample rule.

Another reason is the ease with which the system can be extended and interfaced with C++ , which is the main language used in TDAQ development. C++ extensions are used to interface the CLIPS environment with other parts of the system, such as IPC, ERS, DVS, etc.

The CLIPS system consists of a knowledge base and an inference engine. The knowledge base consists of a number of text files where expert knowledge can be encoded as IF-THEN rules. CLIPS also supports procedural and object-oriented programming, which is used to build a class hierarchy representing proxies of the applications and hardware in the system. These proxy objects are created dynamically using the information from the configuration database, and the information about the objects can then be used in the recovery rules.

CLIPS parses the knowledge base at runtime. This allows the behavior of the ER unit to be easily customized by supplying different sets of files, which describe the knowledge base and rules, as arguments to the application. Fig. 4 shows an example of a simple knowledge-base rule. This simple rule detects the case where an application has died, and it notifies the associated supervisor and the controller to ignore the application from now on.

We will now look in detail at the different steps of the recovery procedure and how each step is handled and implemented in the ER system.

1) Error Detection: The recovery system itself does not do any direct error detection. It relies on the applications themselves and the controllers to notify whenever an error occurs. However, the recovery system might perform tests as a consequence of reported errors. These tests may then discover problems that have not been reported directly.

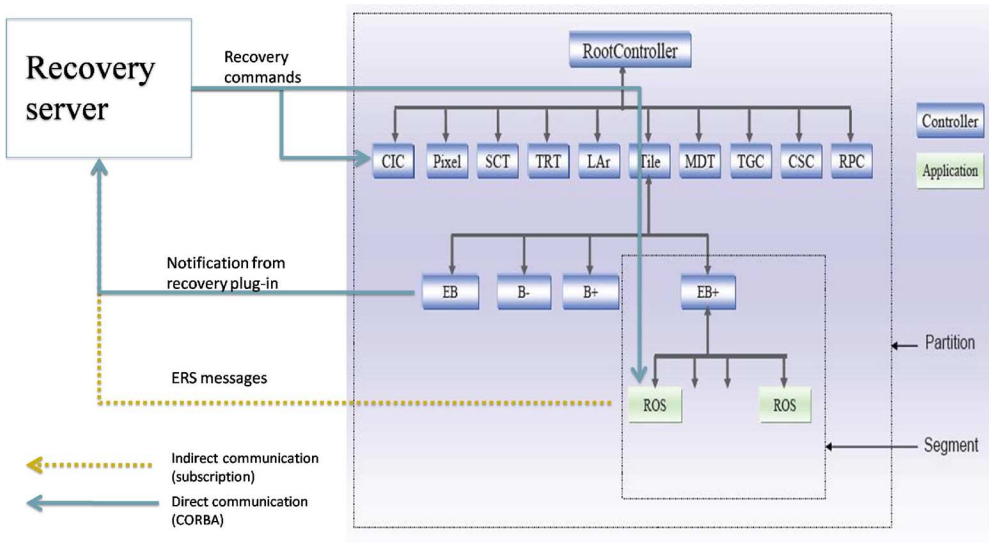

Fig. 5. Overview of interaction with the global recovery server.

2) Error Reporting: Applications report errors to the recovery system using the ERS. This system is being used for all error reporting in the TDAQ system, and there is therefore no specific need for changes to be made for these applications. However, to ensure that all errors are sent using the same format, a set of error classes is defined to be used for specific types of errors. The recovery system (both local and global) subscribes to a set of these classes, depending on its configuration. The applications reporting the error do not need to have any direct knowledge of the recovery system but merely reports the error using one of the predefined classes. The error can then be picked up anywhere in the system and handled appropriately. Note that the recovery units in different segments might subscribe to different classes of problems and that some classes are only handled by the global unit and will never be picked up by the local units, and *vise versa.* One example is a faulty front-end driver. This is known to have impact on several parts of the system and is therefore only picked up by the global server. Errors can also be reported from the local units directly to the global one using the IPC framework. Fig. 5 shows how the global recovery server interacts with the system as a whole.

In addition to the messages reported through ERS, the local unit receives error notifications directly from the controller. This includes notifications such as dead applications (in that segment), errors, timeouts, etc. It also receives updates about the state of the applications and the controller and can use this information to decide on what action to take.

3) Error Analysis: There are two main types of errors that can occur in the TDAQ system. The first type is transient, meaning that the error will not necessarily persist in the system if a full restart is performed. Most software errors/failures are of this type. The other type consists of non-transient errors which will be present even if the system is restarted. Most hardware errors are non-transient and therefore need to be dealt with differently. In many cases, it is likely that hardware errors will not be reported but might cause a software error instead (e.g., an application cannot contact a machine due to a malfunctioning cable). It is therefore important that the recovery system is able to recognize as many of these cases as possible. The DVS is very useful in this respect, allowing the recovery system to automatically test a piece of hardware as a consequence of certain error reports. Also, the configuration database provides some information about how to deal with different errors. The configuration database defines a default behavior in case of an application dying, going into error, etc. However, this is clearly limited as decisions need to be taken based on dynamic parameters such as system state and other errors.

One of the most difficult challenges is handling situations where one error leads to an error in a different part of the system, and so on, creating an "avalanche" of errors. To solve this, it is important to classify which applications are depended upon by others and are likely to cause avalanche errors. One should then try to identify which specific applications will send errors in this case. In this regard, the configuration database is extremely important as it must be possible to automatically retrieve information about how the applications are interconnected, what hardware is being used by which applications, and so on. Even though a human expert might have the knowledge and experience to track down the real error, it is not trivial for a computer program to do the same. It is therefore important to gather knowledge about the consequence of different errors to be able to build a system that will be as effective as possible.

The ER system is able to recognize cases where the system is likely to be non-functional and no recovery is possible. In this case, the run should be stopped, but the general policy is that there should not be a means whereby the execution of the system will be stopped automatically. Hence, the recovery system will therefore need to notify a human operator who can then make the final decision.

4) Recovery Actions: Both the local and the global recovery unit can send commands to all applications in the system through IPC. All controllers and applications share the same command interface, which helps to simplify the recovery procedures. The most important commands are *enable* and *disable*. These commands are used to notify affected components as to whether an application is operational and whether it should

be a part of the system or not. For example, if a processing application in the trigger system is malfunctioning, the recovery system can tell its supervisor and its controller to ignore it using the disable command, effectively removing the application from the system. It can then try to restart the application and bring it back to the correct state. If the recovery actions are successful, the ER can notify the supervisor and controller again, this time to enable it.

In the case of a non-transient error, the recovery system can, if appropriate, make changes to the configuration database. For example, if a connection is known to be malfunctioning, it can be disabled in the database so that at the next reconfiguration of the system, applications will not use it. This type of action is available in addition to the standard recovery procedures.

V. Gathering Knowledge

Gathering knowledge is a crucial point for all expert systems. A large number of developers are involved in the TDAQ project making the task of gathering knowledge even more demanding. As a first step, the ideas for the new recovery mechanism, and especially the reporting mechanisms, have been presented at several ATLAS and TDAQ meetings and conferences in order to make sure that the sub-system developers are aware of the system.

Due to the size and complexity of the system, recovery analysis and actions must be customized for each sub-system. The knowledge of what errors might occur and what actions to take are often held by the individual sub-system experts. A series of discussions have been held with some of the sub-system detector experts to try do define the most effective recovery behavior possible. Even though the knowledge base is written in a human-readable language, it is still too complex for a regular user to modify it. All changes to the knowledge base should be done by a system expert.

As for any computer program, it is very difficult to prove its correctness. This is extra complex with rule-based programs where the execution is done by the inference engine. Testing and feedback from users is therefore an invaluable means by which to both ensure that the system behaves as intended and to further develop the knowledge base. The system is regularly run on test-beds using some standard configurations. Additions to the knowledge base are tested using these setups before being made available for use in the experiment.

VI. Performance and Scalability

In a system of the size of TDAQ, scalability can be a real issue. As the local recovery units are mapped to each controller, they will scale with the control tree as the system grows. The global unit, on the other hand, is implemented using a single instance per partition. However, as the recovery unit deals with errors, there is a natural limit to the number of messages that need to be processed. If error messages are too frequent, it is unlikely that any recovery is possible. Performance is still an issue as the recovery mechanism should be able to analyze the errors and reach a decision within reasonable time span, that is, within a few seconds after the error has been reported. The longer these decisions take, the longer the TDAQ system might be nonoperational. In addition, the likelihood of an error causing an avalanche throughout the system increases as time goes by. This may be due to timeouts on connections or multiple requests to the component in error, and so on. Also, as the knowledge base grows, the execution time of the CLIPS inference engine will increase. Regular performance tests will therefore be performed to ensure that an acceptable level of performance can be maintained.

If performance proves to be a problem, the global unit can be divided into several units which perform dedicated tasks. This will both reduce the number of errors to be handled by each unit and also reduce the size of the knowledge base and, hence, increase performance. However, one must be careful to do this for well-separated tasks, or one would lose the benefit of having a server with a global view of the system.

VII. Conclusions

While the dynamic error-recovery system is still in the early stages of development, the current experience of the system is positive. Several sub-systems have started defining recovery scenarios, and customized recovery units have been designed. It is already possible to keep the system running in cases where it previously would have had to be stopped or manually fixed by an operator.

The extendible design and the use of a knowledge-based system means we are able to add or change the behavior of the recovery system in a simple way without affecting other components. This is very important, especially since many of the error scenarios have yet to be identified or documented, and as feedback is received, additions, modifications, and improvements must be made.

VIII. Further Work

The error recovery system is a work in progress. Focus over the next year will be concerned with expanding functionality to include more recovery scenarios for dealing with failures in all the sub-systems. As feedback is received, the system will be modified and improved, and new rules will be added to the knowledge base. There is also an ongoing investigation to try to identify related technologies which may play a significant role in the development of the system. For example, fuzzy logic is being investigated as a possible means via which to deal with "unknown" errors and other such situations.

Acknowledgment

The authors would like to thank their colleagues with ATLAS TDAQ for their continuing input, ideas, and feedback throughout the course of the design and development of the system.

References

[1] Atlas High Level Trigger, Data Acquisition and Controls Technical Design Rep. ATLAS Collaboration, 2003.

[2] *OMG, CORBA*, [Online]. Available: http://www.omg.org/corba

[3] V. K. A. Poy, J. Sloper, and G. Miotto, "Operation of the atlas experiment: Organization of the detector controls and the data acquisition system," in *Proc. 32nd Annual Conf. IEEE Ind. Electron Soc. IECON*, Paris, France, Nov. 2006, pp. 190–194.

[4] A. Kazarov, A. Corso-Radu, G. Lehmann-Miotto, J. Sloper, and Y. Ryabov, "A rule-based verification and control framework in atlas trigger-daq," *IEEE Trans. Nuclear Sci.*, vol. 54, pp. 604–608, 2007.

[5] *CLIPS: A Tool for Building Expert Systems*, [Online]. Available: http://www.ghg.net/clips/CLIPS.html

[6] D. Liko *et al.*, "Control in the ATLAS TDAQ system," in *Proc. Comput. High Energy Phys. Nuclear Phys.*, Interlaken, Switzerland, Sep.–Oct. 2004, p. 159.